BETWEEN PROGRAMMING LANGUAGES

Toward Solutions to Problems of Diversity

A thesis

submitted in partial fulfilment

of the requirements for the Degree

of

Doctor of Philosophy in Computer Science

in the

University of Canterbury

by

Robert Lewis Biddle

University of Canterbury

1987

Preface

I began this project with a great deal of idealism, having decided that there was a topic in Computer Science that interested me enough to spend considerable time in research. This topic, programming language diversity, has served me well - perhaps too well for the scope of the project. The aspects that interest me most have lead me away from programming languages, to the contexts of wider language and art. In those contexts, the problems of programming language diversity do not admit any straightforward solution, perhaps not any solution at all. This does not displease me, and I look forward to exploring this subject in general, free from concern with "solution".

In practical computer programming, however, there do remain problems of incompatibility in the diversity of programming languages. Accordingly, I have studied the practical methods that have addressed these problems, and concluded that while most methods are successful in limited ways, there is a more general approach suggested. This new approach, "interprogramming", is a collaboration between programming languages, relying on operating systems support. I have described how this collaboration could work, and investigated its possibilities and implications with several programming languages and operating systems. I think it's a good idea: not startlingly dramatic, but a helpful integration of techniques from several areas.

I end the project with idealism still, but weary. I believe the topic is an important one, though little directly discussed, and hope my surveying and exploration worthwhile. More work does need to be done: more theory, more development, more practice. But the topic is very large, and the directions to pursue are themselves divergent. I feel my perspective has become rather lost, and I look forward very much to finding a new one.

I would like to express my appreciation to those in whose companionship I have worked on this project: my supervisor, Bruce McKenzie, and all the staff and students of the Computer Science department at the University of Canterbury. I'd particularly like to thank the postgraduate students of the department for many stimulating discussions and shared explorations of computing, and for their general enthusiasm and devotion to the subject: I wish them well. I am also very grateful to those responsible for the Commonwealth Scholarship that brought me to New Zealand, now my home.

Projects like this are sometimes dedicated to those personally close, but I have always found that inappropriate. I dedicate this project to other naive students interested in pursuing the same topic: I hope my work maps out the area, shows where many directions lead, and helps them understand the difficulties and opportunities concealed.

To my family, to Anthea, I dedicate: new time, new energy, new life.

Robert

ABSTRACT

The success of programming language design is so great and diverse that the resulting incompatibilities have become an important practical problem. This is a study of possible solutions.

The first part is a consideration of reasoning to convergence. An operational model of programming language is used, and both implementation and application aspects are examined, in application encompassing approaches of linguistic precedent, mathematical theory, psychological principle, and empirical analysis. Some progress seems evident in looking at language implementation, but in language application the difficulty of understanding the human factor seriously impedes further reasoning.

The second part is a survey of practical methods for coping with programming language diversity, limiting the diversity or limiting the effects of the problem. Two general approaches are explored, compromise in new design and cooperation in accepting existing design, each comprising a variety of techniques. While no one method was seen completely successful, all had specific advantages and some, particularly cooperative methods, seemed open to further development.

The third part is a proposal for a collaborative strategy here called "interprogramming". In this approach it is suggested that programming language design include facilities here called "context openings" for general connection and communication beyond the language. Supporting environments such as operating systems would need to provide facilities realising such connection and communication. At the programming language level, this requires precise control over input/output and related concerns. At the operating system level, this requires multitasking with memory protection and message passing, or similar capabilities. Exploration of the approach is made involving several programming languages and two operating systems, and some related new software is presented. Most programming languages seem accepting, and operating systems design capable. The strategy should be practical and portable, adaptable with existing design and helpful in new design. As far as programming language diversity is seen as an technical problem, the interprogramming strategy is seen as a general and promising approach to solution.

CONTENTS

LIST OF FIGURES

CHAPTER I
INTRODUCTION

Programming language design is generally thought to have proven a useful and successful pursuit in support of practical computing. While apparently easing some problems, however, continuing design has itself worsened others. In practice, design requires implementation, and the continuing variety in computing facilities has thus lead to interest in portability, both in implementation and in design. But beyond this concern is another, for the diversity of programming language design is itself a practical problem.

Programming language diversity is a problem when one language alone is restrictive. There are problems of availability: when a necessary or useful programme or programme component already exists - but in an incompatible language. There are problems of applicability: when an application spans more than any one programming language context or paradigm. Both reduce programming flexibility and mobility, and thus in turn affect the practical acceptance and experience of design evolution.

Sammet's annual "roster" of programming languages included only languages in use by people other than the implementors, and the number surveyed rose from 42 in the first roster in 1967 to 166 in the last in 1978. As even different dialects of the same language are often incompatible, however, the diversity problem is greater than those numbers suggest. And the number will be larger now anyway, if only because of the more widespread use of language building software.

Diversity is a common problem, a practical problem; it is a result of success in creating many useful programming languages - but one that limits success in using them. As programming language design continues to develop, perhaps in quality and certainly in quantity, and as implementation becomes better understood, so increases the diversity problem. As programming language portability continues to be better understood and effected, so increases the importance of the diversity problem. There has been very little direct attention to the problem in general, though much research and practice in many fields appears relevant.

Winograd (1979) has speculated about programming "beyond programming languages", suggesting computing should follow not from "how", but rather from "what", the programme specifications describe. While the direction is certainly interesting, it is not beyond "language", and only beyond "programming" in the restricted sense of navigation. The word has been extended in application to computing already, and not with such restriction. Anyway, in here speculating about programming "between programming languages" no similar restriction is implied. This study is an exploration of possible solutions to the problems of programming language diversity.

PERSONAL MOTIVATION

I have long been interested in the expressive power of programming language: since the confrontation between my first learned programming language, APL, and my second, Fortran. In his consideration of the consequences of one's first programming language, Wexelblat (1981) places APL in the "strange" language category, and is concerned about the difficulty people like me must face in moving to

"conventional" languages. I have moved with strong, though often ignorant, partisan support of favourite languages, like APL, through daily practical application of many languages. But as practical involvement with differing languages and paradigms increased, so partisan feeling diminished. I know, as I believe all practical programmers know, language design can and does affect programme and programming quality. But quality, of programme, programming, and so of programming language seems an uncertain basis on which to reason further. And yet, the resulting and remaining diversity is itself a problem.

As an artist must consider both canvas and palette, so the programmer might then consider both programme and language. And as next beyond quality concern is portability concern, so even then beyond is diversity concern. In an early essay I considered the palette problem of representing arbitrary data structures in general purpose languages (1983a). I suggested a "relational" approach in contrast to the usual "network" orientation, further showed that implementation efficiency would be comparable, and as an example suggested a modification to Pascal. But there the question of the importance of the diversity concern became alarming. Certainly most, perhaps all, programming contexts include some bothersome aspects. And it is not too great a difficulty to conceive new language facets or indeed new languages; it's rather fun. But how to evaluate their worth without practice?

```
type    refperson = ^person;
        person = record   name: nametype; sex:  sextype; age:  agetype;
                          nextyounger, nextolder:   refperson end        {Old}
var     nextyounger, nextolder: mapping[refperson] of refperson;       {New}
...
{Old}   pprevious^.nextolder := newperson; newperson^.nextolder := p;
        p^.nextyounger := newperson; newperson^.nextyounger := pprevious

{New}   nextolder[pprevious] := newperson; nextolder[newperson] := p;
        nextyounger[p] := newperson; nextyounger[newperson] := pprevious
```

Figure 1.1.1: An alternative in Pascal data structure notation - It would be possible to replace explicit pointer operations with a "mapping" structure of the same efficiency. This might be argued notationally superior, but it is not clear how to resolve such issues. Where such new ideas are explored, potentially bothersome incompatibility can result.

In practice both problems with portability and diversity dissuade investment in new ideas - even in their exploration. Accordingly, great efforts have been made in programming language portability, and with useful success. With such success in portability, however, consideration and understanding of the diversity problem seems to me imperative. Moreover, while the programming language diversity problem is directly one of practical computing, because systems of notation and other representational schemes are so widespread the computing problem is closely related to questions in many other fields, and in wider general experience. So the exploration of the diversity issue provides an opportunity for consideration of the relationship between computing and other experience, a subject to which I personally find myself continually drawn. And, to say so explicitly, the reason for my perseverance in spelling "programme".

PERSPECTIVE

The problems of programming language diversity do directly involve practical, operational, programming. An operational orientation seems thus most suitable: concern with the actual, though general, methods of programming practice. So saying is to exclude direct concern with theoretical programming - indeed the diversity problem as described does not there arise (in as much as there is a distinction between theory and practice in as abstract a subject as programming). The viewpoint of most direct concern is thus that of the programmer, albeit the programmer in general. So saying is to exclude direct concern with others less directly concerned in operational programming - computer manufacturers or marketing representatives may well see diversity, and portability, differently.

But while essentially practical in purpose, discussion of the diversity problem must be at a high level of abstraction so to cover the understood variance in actual practice. Some even common nomenclature used in the discussion is not well agreed upon, so some declaration of usage here is called for. "Computing" is the formal mathematics of symbol manipulation of concern, and more generally the field surrounding and based upon it. A "computing model" is a formal mathematical context for computing, a "computer" is an implementation of a computing model. A "programme" is a description of computing within some computing model, and "Programming" is the human activity involving expression of a programme or programmes. A "programming language" is a computing model for direct human expression of programmes. In this way "programming" is a more general term than that implied by Winograd.

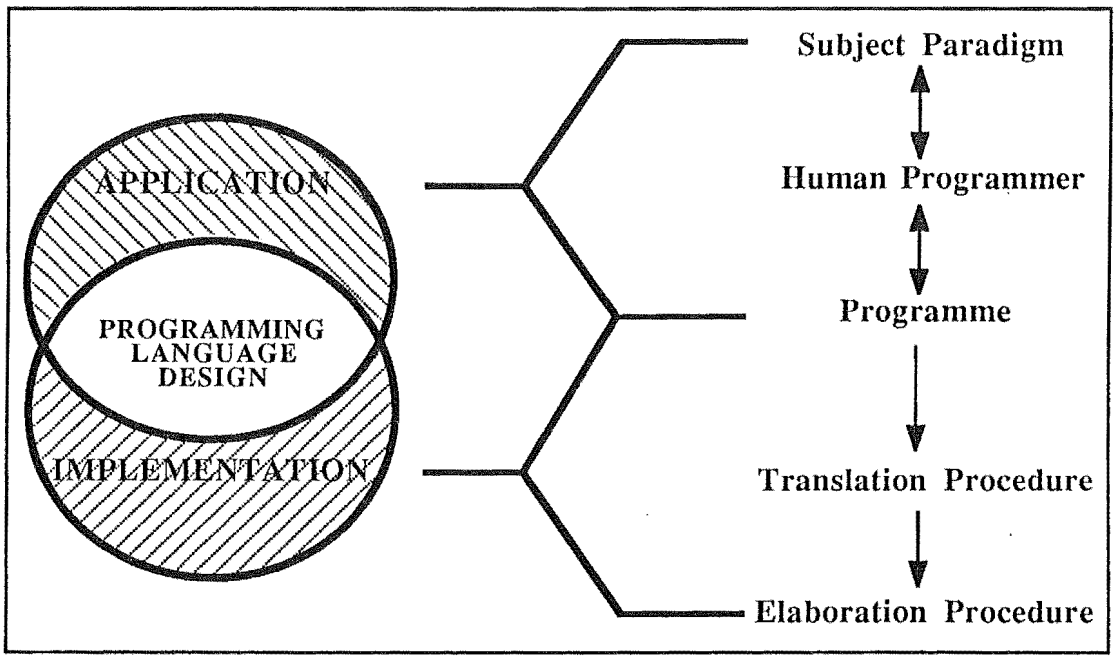

Figure 1.2.1: Structural relationships in the terminology used -"Programming" is distinguished as the necessarily human activity of programming language application; translation is distinguished as the separate concern of implementation. These two aspects meet in programming language design.

However, in discussing the diversity problem, concern is mostly with longer lived programmes as reusable artefacts. Accordingly, "programme" and "programming" will usually concern these artefacts and not the more transient and often interactive case. In some situations the distinction of interactivity is difficult anyway. Few limits are intended on interpretation - inclusion of canonical practice is certainly implicit.

It is, however, the artefact nature of the programme and the mathematical context defining its operation that is the kernel of the particularly practical nature of the diversity problem. Computing, and programming, necessarily do not have the flexibility of many other notational contexts: for practical computing that is a strength so novel for mathematical models, though here it is also a weakness. Hence a distinction between computing and programming, and hence a distinction between implementation and application - the practical process and practical artefact are of primary importance in discussing diversity.

Even in the operational practice involving programming language, however, two aspects are seen: application, language and programmer; and implementation, language and (eventually and recursively) computer. In application, the operational activities of writing and reading are day-to-day familiarities. In implementation, the operational activity is translation, interpretation or compilation as in linguistics. All these distinctions can then lead to a resolution of the nature of the diversity problem, the first and necessary step in understanding how the problem might be resolved.

PLAN

Over all, this exploration of solutions to the problems of programming language diversity is structured with three divisions progressively narrowing in scope.

First is a consideration of the possibilities for reasoning to convergence in programming languages, thereby eliminating the problems of diversity by eliminating the diversity: integration at the roots of diversity. Central here is the question of programming language "quality" and how it might be evaluated. These questions are extraordinarily deep, and conclusions can really only underline the difficulty and importance of the problem. In these discussions scope is generally restricted to matters directly involving computing. As matters seem to lead away to other experience and more general enquiry, this is somewhat arbitrary and perhaps rather unsatisfying. Within the practical context, however, simply establishing the direction of difficulties is sufficient.

Second is a survey of established computing practices and techniques that in some way approach solution to the diversity problems. This a reminder that much that is impossible to totally resolve can still managed with tolerably. The approaches examined are grouped into those seeking "compromise": reducing different languages to one, and those seeking "cooperation": ways of using different languages together. Often the diversity problem has been only a minor or unstated concern of formal developments and studies, so some conclusions can be only informally drawn. Many strategies still have definite advantages in certain circumstances, however, and several deserve to continue being applied. But most strategies also have

significant drawbacks that do keep their usefulness limited. Even so, some cooperative practices appear on examination to suggest further development of a more general strategy may be worthwhile.

Third is the proposal for a new general strategy for addressing and tackling the diversity problem. The philosophical problems realised in the first part of the study and the practical directions suggested by the second section do not quite seem in collision. Indeed it seems possible to practically evade problems of diversity as much as any particular programming language design permits, while avoiding any necessary implication of impossible universality. So such success must necessarily be limited, but some increased success is possible. The approach involves minimally collaborative design in programming languages, and necessary support from environments exterior to them. Though grounded in applicable practice, these proposals do not follow through in requiring specific action - there remains much choice. These are design principles for languages and support systems, and could be evolved toward. However, several programming languages are explored, and opportunities for useful access described or modifications suggested. To practically examine the question of the exterior support, two operating systems are also discussed, and the requirements of necessary design established. The importance of new software both to assist and take advantage of this "programming between programming languages" is illustrated with some new programming tools.

So more exploration and experience are indicated, and detailed study for particular programming languages and operating systems design would then be of most concern, as well as more attention to particular questions of performance and efficiency. Such particular work is outside the current scope, as indeed are several other directions suggested for future investigation. Explorations in representation diversity in other fields may deserve investigation next, and the balance of subject matter could be important as well as enticing. The contributions of this study are the mapping out of the vast area of this important subject, the review of relevant past practice, the development of a general strategy for progress, and the directions for its practical pursuit.

PART ONE

CONVERGING

From INTRODUCTION:

Over all, this exploration of solutions to the problems of programming language diversity is structured with three divisions progressively narrowing in scope.

First is a consideration of the possibilities for reasoning to convergence in programming languages, thereby eliminating the problems of diversity by eliminating the diversity: integration at the roots of diversity. Central here is the question of programming language "quality" and how it might be evaluated. These questions are extraordinarily deep, and conclusions can really only underline the difficulty and importance of the problem. In these discussions scope is generally restricted to matters directly involving computing. As matters seem to lead away to other experience and more general enquiry, this is somewhat arbitrary and perhaps rather unsatisfying. Within the practical context, however, simply establishing the direction of difficulties is sufficient.

Second is a survey of established computing practices and techniques that in some way approach solution to the diversity problems. This a reminder that much that is impossible to totally resolve can still managed with tolerably. The approaches examined are grouped into those seeking "compromise": reducing different languages to one, and those seeking "cooperation": ways of using different languages together. Often the diversity problem has been only a minor or unstated concern of formal developments and studies, so some conclusions can be only informally drawn. Many strategies still have definite advantages in certain circumstances, however, and several deserve to continue being applied. But most strategies also have significant drawbacks that do keep their usefulness limited. Even so, some cooperative practices appear on examination to suggest further development of a more general strategy may be worthwhile.

Third is the proposal for a new general strategy for addressing and tackling the diversity problem. The philosophical problems realised in the first part of the study and the practical directions suggested by the second section do not quite seem in collision. Indeed it seems possible to practically evade problems of diversity as much as any particular programming language design permits, while avoiding any necessary implication of impossible universality. So such success must necessarily be limited, but some increased success is possible. The approach involves minimally collaborative design in programming languages, and necessary support from environments exterior to them. Though grounded in applicable practice, these proposals do not follow through in requiring specific action - there remains much choice. These are design principles for languages and support systems, and could be evolved toward. However, several programming languages are explored, and opportunities for useful access described or modifications suggested. To practically examine the question of the exterior support, two operating systems are also discussed, and the requirements of necessary design established. The importance of new software both to assist and take advantage of this "programming between programming languages" is illustrated with some new programming tools.

So more exploration and experience are indicated, and detailed study for particular programming languages and operating systems design would then be of most concern, as well as more attention to particular questions of performance and efficiency. Such particular work is outside the current scope, as indeed are several other directions suggested for future investigation. Explorations in representation diversity in other fields may deserve investigation next, and the balance of subject matter could be important as well as enticing. The contributions of this study are the mapping out of the vast area of this important subject, the review of relevant past practice, the development of a general strategy for progress, and the directions for its practical pursuit.

CHAPTER II
CAN WE REASON TO CONVERGENCE?

Programming language diversity has arisen amid the novelty of the application of computers in general, and to many particular fields in turn, by a community itself diverse in background and in outlook. In this time, however, much has been learned about programming, and about programming language. The rare explicit discussions of the programming diversity problem generally simply accept that re-convergence will not occur (see Beech, 1982, for example). Perhaps it is time to question this, and examine the possibilities for reasoning to convergence in the now more mature discipline of computing. Reasoning to convergence will require selecting few languages from many, so will require comparing languages. Comparing languages involves many factors, but must centre on the effect of the language itself on the programming: the language "quality". But any argument for convergence must not only show direction, but must so do with great confidence. The weight of past knowledge, learning, reasoning, and expression will shun any proposal unlikely to lead to convergence, or unlikely to lead sufficiently far. Convergence may seem attractive: the possibility of its realisation needs be convincingly strong.

Programming language diversity was fostered not only by difference in perception of quality, but by practical and pragmatic expedience. Much of the pragmatic problem concerns simple availability. In essence, this is the problem tackled through seeking language implementation "portability", and has received much attention resulting in several approaches of notable success and developing promise. The problems were recognised early and have lead to the very successful use in well known languages, the general "macro" implementation for Snobol (Griswold, 1972), translation to generalised target languages for BCPL (Richards and Whitby-Strevens, 1980) and Pascal (Pemberton and Daniels, 1982), and heuristic "re-targeting" strategies: Ganapathi, Fischer, and Hennessy (1982) provide a survey. Much work in understanding and applying formal programming language specification promises to even further lift problems of availability (see Cattel, 1980). There does seem to be significant progress being made, and some implications of this are discussed later.

In practice, of course, pragmatic influences may be imperative or irresistible, for good or ill. Typewriter keyboards are almost standard, for example, for ill, as Montgomery (1982) discusses while reviewing new alternatives. And Gries (1981) suggests Fortran is the equivalent to the standard "qwerty" keyboard. Fortunately, though, there is hope yet that programming language in general may escape that fate, or others worse. But the quality problem remains dauntingly formidable. There is clearly strong feeling now as ever on a variety of aspects of the problem, but it is not immediately clear whether such feelings can be encompassed in theory, explored, or verified. While the difficulty can seem beyond superlatives, the study of the problem has attracted considerable attention in recent years, and indeed always at least indirect attention. Operational experience structures programming language about implementation and application, a start in grasping some order. Of course many pragmatic and external influences would affect any possibilities of convergence, but this influence is not under consideration here. What is under consideration is the possibility for reasoning to convergence on grounds of programming itself, and so a look at the different approaches to language quality in those terms should prove indicative of what might be achievable.

SECTION I. IN IMPLEMENTATION

In examining language quality from the implementation perspective, the practical importance of the translation process is paramount. In this translation, only computing is regarded as significant. As both source and target are formal computing models, translation can follow formal analysis and synthesis. In analysis, two criteria seem significant: complexity of the translation as primary; generality of that complexity as secondary. And complexity is of great practical importance because it consumes resources valued in practice. In isolation, though, the "complexity of translation" means little without reference to some target model. This is the starting point of the paradigm for research in algorithm complexity since established by Hartmanis and Stearns in 1965 with the basis of the "abstract machine". While mathematically several models might be of interest, pragmatically it is those ones similar to those that can actually be physically constructed that are regarded significant. So with that generality considered, "complexity" refers implicitly to some sufficiently general and common target computing model. Recent movements in complexity theory have in fact focussed on the probabilistic orientation suggested by Karp (see Karp, 1986) because of the difficulties in lack of distinction and the often practical inappropriateness of worst-case analysis. It may be that a similar line could be taken in programming language analysis yielding more useful distinction in language design, but the avenue appears yet unexplored.

Complexity defers to computing power, though in practice this is not important: programming languages generally demand all the computing power there is and no more. Complexity defers to completeness and consistency too, as otherwise resulting syntactic and semantic ambiguity can limit language and programme generality and hence portability. Ambiguity resulting from informality in specification of Fortran (USA Standards Institute, 1966) for example, resulted in the later need for specific clarifying documents (1969, 1971), a need that continues. Emphasis on formal language specification, however, has greatly lessened this implementation nightmare.

Translation complexity clearly is formally limited by source and target languages themselves, but as both are in some circumstances plastic, opportunities do arise to effect language change in order to increase efficiency. Translation complexity has been seen as an important measure of overall language design, originally overwhelmingly so (see Backus, 1978b, for examples in the design of Fortran), and programming languages are usually well scrutinised for unwarranted translation demands. More formally, much investigation has sought language characteristics that best allowed efficient translation methods.

Both lines of enquiry have been fruitful. Informally, it has been realised that many tiny details of language syntax seem of little concern to programming language users. Formally, the developing "formal language theory" showed that some language designs allow parsing procedures of low complexity, and others do not. Accordingly, languages evolved to slightly different syntax structures that enabled superior implementation procedures. This evolution has lead to a convergence of syntactic structure in most modern programming languages: small patterns of limited regular languages strung upon the context-free backbone of a general phrase structure language. The context-free backbone is often seen as the most significant structure,

and commonly involves, more specifically, LL languages that permit easily written recursive descent parsing, or LR languages that allow the application of automatic parser generators. The line of study continues, the current work exploring not so much better parsing methods, but rather strategies to allow complete automatic implementation from formal specifications (Cattel, 1980). Clearly such definition is possible, as the two level grammars of van Wijngaarden (1965) and used in the definition of Algol 68 (et al., 1969) have been shown (Sintzoff, 1967) capable of generating any recursively enumerable set. This includes not only the total syntax (including static semantics) as in the revised report (van Wijngaarden et al., 1976), but also the total language (including dynamic semantics), as demonstrated by Cleaveland and Uzgalis (1977). Of course, this has linguistic problems too, as such formal specifications may not be particularly suitable for human use. Language definition could then proceed on such a base, however, as Pagan suggests (1975, 1978, 1980b), using a language such as Algol 68 itself for definition. But it seems doubtful that any approach, no matter how seemingly formal, can be entirely rid of all informality. Even in recursive definitions, such as that for simple Lisp (such as given by McCarthy, 1978a), or in mathematical definitions such as those following Scott and Strachey (1971, and Stoy, 1977), there seems a difficulty inherent in human application. While the human component is not formally understood - itself a problem with interesting recursive implications - formality in language specification cannot be total, and cannot determine convergence.

But while the human element is inescapable, our experience is that some formality has helped. So in language implementation anyway, some convergence in ideas of language quality seems to be developing. The impetus has been the desire for demonstrably efficient translation, and has been met by development of linguistic models that seem to permit minimally complex translation while demanding little. And so leading to some narrowing in language structure.

SECTION II. IN APPLICATION

While the human mind is not new, and while our experience with it is extensive as any we have, understanding of its behaviour is limited and unclear. This may seem trite, but it is seems seldom mentioned and oft forgotten in much design. In programming language application, the human factor is central - "programming" languages were originally developed to ease the human task of machine programming. In analysis of the effects of language design on language application several approaches are popular: deference to linguistic precedence, mathematical consideration of the notation, and psychological consideration of the programmer.

Even distinguishing the human activities involved in language application illustrates the problems faced throughout. The primary direct activity is clearly expression - writing. Chronologically consequent, but in many cases more frequently involved is comprehension - reading. There are several intertwinings, though: surely some reading is involved in any sufficiently lengthy writing; and perhaps "maintenance" might involve much reading followed by little writing. Much thought has recently been given to formal reasoning about programmes - is this merely "comprehension"? And the problems go deeper still: for programming and programming languages have themselves become involved beyond initial intent. On a low level, for instance, it is often possible to "test" or "de-bug", exploring dynamic behaviour. On a high level, once the notation exists and is sufficiently familiar, it may itself augment or supplant other notations in use in the application paradigm and become itself a conceptual framework. Is "thinking" a language application activity? It seems so, at least to some extent, and analysis is then particularly difficult.

In evaluating language design in service of these activities we might look first, as with implementation, to complexity and its proviso, generality. However, we lack formal models for human behaviour and notice moreover that, in detail and in general, humans err. Should other values be considered: robustness? reliability? safety? It's difficult to argue otherwise.

Certainly evaluation is beyond any simple axiomatic approach. Many axioms we do have themselves conflict, and resolution is a human concern. For example, consider compactness: to begin with, the converse, redundancy, is valued too; to finish with, the reality of serious compactness is worth remembering. Even in related programming contexts, severe disagreements arise between experts. In comparing two languages of a similar age and popularity, C and Pascal, both Algol 60 descendants, both general purpose programming languages, several studies disagree. Several direct comparisons, for instance, most notably that of Feuer and Gehani (1979, 1980, later 1982), prompted Kernighan to explain "Why Pascal Is Not My Favorite Language" (1981) relating experience of remaking the Ratfor (Fortran preprocessor) based Software Tools book (Kernighan and Plauger, 1976) to a Pascal base (1981). He was concerned that as a language for practical programmes that would work together, Pascal had important deficiencies, and much effort was expended in overcoming them. Then to face reviews long and scathing in their condemnation of the approach. Lecarme (1983) concludes:

"... they do not use any of the new capabilities provided by a language which is new for them (although more than ten years old). On the contrary, they force the language to conform to their habits. The result is a book I consider to be, despite indisputable qualities, hampered by major defects that make its publishing an error. The worst thing is that the authors seem to be sure of holding the only truth. They force

upon the reader their programming tricks and manias, and fill their text with self-satisfied remarks in the style of 'Consider what we do, and proceed like us.'"

So experts differ.

Surveys of programming language, if considering evaluation at all, usually discuss and illustrate how different values yield different languages. Clearly most languages are designed to fit some frame, and in such terms evaluation might be possible. But there are many such frames, and so arguments assuming some understood universality, such as Iverson's (1980), assuming great importance for lexical compactness (for which Iverson's own APL design has become famous or infamous) are not convincing. What method might lead to resolution?

Linguistic Precedence

In the history of programming languages, probably the earliest value seen significant was that of correspondence to another notational model perceived important. Fair enough, computing is abstraction: one can know nothing of computing without knowing anything of something else. Notational precedence is an attractive approach in that it avoids directly addressing new questions of value. Pragmatically, programming language graphemes have most often been drawn from symbols already in use, most often in commercial business practice. Nicholls (1975) in discussing this lexical level refers to the mathematical history of symbols compiled by Cajori (1928), but it is clear from work on character set standardisation (see MacKenzie (1980), for example), that recent business machine practice has a great impact. In well established languages probably only APL bucks this influence, and it too (see Iverson, 1962) is itself based in mathematics practice.

Weinberg, in 1971, does say that no work has been done on the human factors in programming languages, but this is perhaps misleading - it was a major goal of, for example, Fortran (Backus, 1978b). Though Backus does say "... we simply made up the language as we went along", the remark is perhaps misleading. Fortran follows notations of numerical mathematics - not coincidentally, its target application paradigm: "engineers and scientists". In later language design, other notations in mathematics and general use have frequently been followed. Recently, for example, widely popular and successful programming systems have been modelled on the business "spreadsheet". The first establishment of the model, VisiCalc (see Ramsdell, 1980), was a reasonably straightforward interactive programme relying on fast access to a screen on the newly economical personal computer. Large modern spreadsheet systems, however, now commonly encompass integrated macroprocessors and verge on programming language (see Poole, 1985). A more recent example of the same phenomenon is the Hypercard programming system of Atkinson (Apple, or Goodman, 1987) where the "card index" paradigm is stretched to programming language status.

Not only business practice paradigms have been applied to programming languages. In "Programming By Rehearsal" (Finzer and Gouls, 1984), for example, object oriented graphics programming is modeled on stage presentation. Individual programmes are referred to as "performers", and several instances of a particular performer is a "troupe". Programming and debugging is done in "rehearsal",

and a successful "performance" of screen graphics is the goal. And perhaps marking some current outer bound of this approach is "Puppy Love" (Snyder, 1987), where a beginners programming language and accompanying development environment is completely oriented about the paradigm of teaching tricks to a (simulated) dog!

Even simple mathematics offers choice: consider the wealth of notation, both in symbol and in structure, employed in common arithmetic. And natural language, of course, embraces an awe-inspiring variety, both in diction and in grammar, employed in common speech. Further, such choices of paradigm may themselves be inter-related: one precedent might seem appropriate from a first point of view, but another clearly more so from a second. There may well be an argument for following precedent; but choosing which precedent to follow may well still be unclear.

In the concluding sections of her 1969 language catalogue, and again in concluding a 1972 retrospective, Sammet looks to the future and discusses the possibilities for English as a programming language. Sammet herself, of course, was involved in the development of Cobol, the early and most successful language designed with a similar goal in mind. If the difficulties of realisation are ignored, the advantage of familiarity of English (to the English familiar) might be clear. For interactive environments, interfaces based on English do now have many advocates (Ledgard, 1981, for example, or Small and Weldon, 1983). Of course, a great wealth even of interactive practice is not so based, yet it is popular. And English-like languages are still being developed for non-interactive use, work such as that by Biermann and Ballard (1980) attempting to use more advanced linguistic principles and a more complete development environment to improve their acceptability.

There have also been related suggestions, and Wile, Balzer, and Goldman (1977) do discuss how conventional programme language structure might be "derived" from English text. A less direct approach is that of Hobbs (1977), who suggests some very general rules from linguistics might be applied to programming language design, redundancy, for instance, ellipsis of assumed components, and anaphora - use of following explanation. And, interestingly, a dominant orientation about spatial concerns. To the comeback of MacLennan (1979), however, that the history of notation is one of departing from natural language because of inadequacies. Notably Hill (1972, 1983) has claimed that the imprecision of English is exactly what needs to be avoided in programming. Plum (1977) warns particularly about the danger of structures that appear similar to ones in natural language, but in detailed operation work differently. Taking this concern to the limit, Beckman (1977) points out that there is even a danger in allowing natural language "comments" in programmes, because inaccuracies there cannot be formally checked, yet have great power in misleading readers. And apart from any disadvantages of natural language, Schneiderman (1980), points out that for many a specific task the precedent most applicable is not any natural language such as English, but rather the specific notations and languages that have become established for specific applications.

Yet notational precedence is not itself always clear. Some programming application areas, and languages addressing those areas, have clearly grown beyond any previously existing notation. In character string processing, for instance, no practical area really existed prior to the effort culminating in Snobol. And when Snobol seemed inadequate, there was no notation to model, and later development concerned a new

model, a new notation, and a new programming language (Griswold, 1978). Curiously, while a number of ideas behind were clearly better thought out in the languages evolving from Snobol, the language structure itself became less distinct. So the current carrier of that torch, Icon (Griswold and Griswold, 1983), resembles much more general purpose languages very closely - to the disappointment of some enthusiasts.

For still, as notation and models in any application area are only human inventions, it is clear new notations and models following a new programming language might perhaps prove superior. Correspondence to an existing notation may seem initially mnemonic, familiar and reassuring, but may otherwise prove a liability.

In defending canonical practice, it appears easy to overstate the case. Quite aside from the symbols involved, APL bucks tradition in having expressions evaluated linearly right to left. Ledgard and Marcotty (1981) remark on this, and quite reasonably point out that "it might well be that the APL rule is the best one and traditional mathematics is wrong in its approach". But they then refer to the traditional approach as "the intuitive one of most of us". This might be, but it is not manifestly obvious, and the person who learns APL first might well express doubt. But the issue is wider than APL. Ledgard and Marcotty themselves quote Whitehead (1911):

"By relieving the brain of all unnecessary work, a good notation sets it free to concentrate on more advanced problems, and in effect increases the mental power of the race. Before the introduction of Arabic notation, multiplication was difficult, and the division of integers called into play the highest mathematical faculties. Probably nothing in the modern world would have more astonished a Greek mathematician than to learn that ... a large proportion of the population of Western Europe could perform the operation of division for the largest numbers. This fact would have seemed to him a sheer impossibility. ... Our modern power of easy reckoning with decimal fractions is the almost miraculous result of the gradual discovery of a perfect notation."

The current western notation used in arithmetic may seem much preferable to others, but "perfect" is a strong word. It seems some distance from the utility of following precedence to any reasoning far to convergence. Clearly it might possibly lead that way if followed, but it's not clear that it's significantly more attractive than other directions.

Mathematical Approach

In consideration of the mathematical characteristics of programming language application, the human factor is largely ignored. No one openly disputes its importance, but many seem to feel it premature or perhaps even even pretentious to build much upon such a flimsy foundation as our current understanding.

A programming language, more importantly than absolutely anything else, is a computing model. The formal power of a computing model is usually expressed in terms of equivalence to one of several well understood models, from finite automaton to Turing machine. While several classes of model are understood, and while the differences in their power are important in several areas in computing, this knowledge yields little illumination about programming languages. Virtually all programming languages are in fact equivalent to each other, and to the Turing machine - the most powerful model known. This knowledge is not very helpful in evaluating languages: we are reminded that we could write operating

systems in Cobol and payroll systems in Lisp - but it still seems silly. Even when some less language-like contexts require less powerful computing models, the comparisons of interest frequently involve similarly powerful alternatives. On the other hand, it is possible in practice for simple languages to model more complex ones sufficiently well for many applications given sufficient resources - most computers are finite state automata, though mostly programmed otherwise.

As discussed briefly with regard to language implementation, consistency and completeness in specification has been much explored and developed since early programming languages. In application too this is important as programmers can find it uncomfortable wandering into undefined or ill defined language. Many such problems seemed reduced as data typing was widely introduced, and formal techniques of language definition and have now done much to further their reduction. The bewildering English of the early Fortran standard was eclipsed by the BNF syntax definition technique employed for Algol 60 - both were surpassed by the "two level grammar" used by van Wijngaarden in defining both the syntax and "static" semantics of Algol 68 (van Wijngaarden et al., 1976). Cleaveland and Uzgalis (1977) showed that "two level grammars" can totally define a language, but the notation is often thought again bewildering. Marcotty, Ledgard, and Bochmann (1976) provide a survey of definition techniques, and McGettrick (1980), and Pagan (1981) more recent texts.

The formal tractability of programming languages and their underlying computing models is a subject of much recent dispute and controversy. Perhaps the best example of this is the question of "Structured Programming", firstly that regarding "control flow". In the classical computer model, "control" is achieved by the loading of a memory address. At machine level, this is the "branch" instruction; at the "higher" level of programming languages, this is the "goto" of languages from Fortran on. Historically, in accordance with a desire to serve frequent needs with well suited facilities, language designers provided statements to simplify the specification of common control statement patterns: most commonly the "while" and "if-then-else" structures. Such constructions proved popular and were widely used.

Interested in the theoretical properties of control statements, Bohm and Jacopini (1966) studied these language structures and concluded that the new control structure yielded the same computing power as the "goto". Dijkstra (1968b) then proposed it was the "goto" itself that was harmful - and that it be wholly abandoned in programming language design. There was widely felt shock. As argued in quick reply, and later more reflectively by Knuth (1974b), there did seem to be some situations, such as the "n-and-a-half times" loop, that seemed to occur rather frequently and required an explicit branch - or some other control structure - for their elegant construction. Denning (1974) suggested abandoning the term "structured programming" altogether because there it was being interpreted as compelling much more rigidity than was warranted. And in their survey of control structure possibilities, Ledgard and Marcotty (1975) trace the histories of several kinds and many variants, each applicable to some particular circumstance.

The idea that the "goto" might be done without and replaced by a small set of higher constructs was debated at length. Many arguments were advanced for abandonment, many informal and perhaps even social. But Dijkstra's argument was itself directly none of these. It was simply that the inclusion of a goto in

a language allowed the existence of programmes about which little could be said, little could be reasoned. Formally, one could not be certain of their behaviour, and thus they could not be tolerated.

The idea was sufficiently successful to markedly change programming, programming language, and programming education. As such, mathematical tractability has proven a powerful influence in language design. One consideration has been whether programmes reliant on explicit branching might be structured automatically. While Bohm and Jacopini have shown this is always formally possible, some more recent work suggests it might further be practically useful (see Baker, 1977) in structuring "unstructured" programmes. In other discussion, it is suggested that models far different from the "von Neumann" might lead to even superior tractability: hence Backus' (1978a) argument for the superiority of "functional" programming languages.

A second major development in programming languages founded on a perhaps mathematical basis is the importance of data "typing" or "strong typing". While there is an important mathematical facet to typing, the influence of "strong typing" as a mathematical reason for programming language convergence is perhaps overrated. The important mathematical facet concerns language definition: the concept of type enables distinction that frees language design and analysis from some implementation concerns while maintaining total definition and ensuring programme integrity. In this way, an implementation may deal differently with, say, a letter of the alphabet and an integer without needing be concerned about their logical disjunction. In the same way, "overloading", of language symbols with different meanings for different type becomes possible.

There are several reasons why typing is not a strong force for convergence. Firstly, it should be admitted that in practice programmers often do wish access, both reception and transmission, to some language implementation levels - typically data storage encodings and control structure generated code. The reasons are usually to do with extending the possiblities of the language, and the efficiency of access is important too. The access at all, and more so efficient access, erode the mathematical argument for typing: where it cannot be ensured itself, it cannot itself ensure.

Secondly, it might be mentioned that typing, strong typing, may be achieved in surprising ways. Should a language be willing to embrace definition of it's own implementation, as many "systems programming languages" might do, even the most "typeless" language might thus be "strongly typed" - even the Turing Machine could be strongly typed. Nor need this be a capability of the lowest level languages, as concern with implementation might also be achieved by an implementation itself flexible enough, interpretive enough, to join the otherwise disjoint.

While that last strategy might achieve the mathematical rigour that typing can offer, the continual interpretive nature may seen burdensome. However, there is a continuum of interpretation required with a continuum of flexibility required with a continuum of surety in integrity. With a great deal of interpretation, comes a great deal of flexibility and of surety. So it is with interpreter oriented languages such as Lisp and Snobol. And with no interpretation comes no flexibility and no surety. And indeed no computing: for some amount of interpretation and flexibility are necessary in any computing model, and some amount of surety is in practice always provided for. So in practice the lower bound is rather higher, but there is always an

implicit trade-off. Pascal, for example, restricts some flexibility in demanding some type surety in programme definition. The rest, and there is always more, becomes an implementation responsibility. What is the error in division by zero, if not one of type?

It remains a possibility that our usual computing model is too difficult for our mathematical tools - the issue raised in the flow control question. If so, we must then decide whether we should change our computing model, or change our mathematical tools. It remains a possibility that mathematics cannot directly help in these matters at all, as Goedel (1931) has shown it cannot always help itself. There may be programmes that are completely correct that cannot be so proven.

Within mathematics there is much diversity yet equivalence: cartesian and polar coordinates do not govern separate applications, nor do different bases govern different arithmetic, nor yet different numeral systems govern different numbers. Do they? In the human application of programming language, is "tractability" a mathematical property? It's difficult rule on such questions because of the wide range of factors involved with the human element: books may not be completely judged by their covers, but covers are associated with books and so cannot be ignored.

At the higher level of discussing programming, the possibility of confusion between mathematics and human politics has caused concern. De Millo, Lipton, and Perlis (1979) expressed concern about the interplay between mathematical reasoning and social argument, especially with regard to the structured programming debate. Interpreting Dijkstra's stand as one for formal verification of programmes, they questioned the role of such a difficult and theoretical process when better operational engineering techniques would be more appropriate. As Dijkstra makes clear, however, especially with Gries (1981), he advocates not verification afterward, but formal development throughout. But the further point of De Millo et al. remains significant: they claim that even mathematically formal proofs - and this would apply to development as well as verification - are subject to social processes before being accepted. The argument basically points out that what is universally accepted true one day, may be proven false the next. As it is possible to make a mistake in a programme, so it is also possible to make a mistake in a "proof". This leaves any superiority claim for a mathematical approach in doubt because it puts mathematical development and operational programming on an apparently even footing. There are further arguments both ways, but no conclusion.

Regardless of the philosophical difficulties, priorities at the day-to-day levels of programming may be also disputed. Languages can certainly be designed on formal principles, as Tennant (1977) has shown, so to assist in the formal analysis of programmes written in such languages. But it is neither guaranteed nor even clear that such languages will make it easy for people to write programmes, or to read programmes, well. Naur (1975), for instance, is happy for theoreticians to follow mathematics, but feels that for applications programming more pragmatic models - from natural language already familiar - apply. And of course there really isn't any resolution, as there is no appropriate context for recourse.

Wittgenstein and Russell immediately agreed that "... in fact all the propositions of logic say the same thing, to wit nothing" (Wittgenstein, 1961). Is there more to programming than mathematics? Rather more probably that there is more to both than formal denotation. Programming is certainly grounded in mathematics, but even in more conventional mathematics, informal importance is attached to "mathematical

elegance". However, the practical and continuing nature of programming as a day-to day activity suggests this must be taken more seriously. The importance of context is clear, but determining relevant context is not straightforward.

Psychological Approach

Psychology is not directly able to answer the questions of language value, but the experimental methods of cognitive psychology have been applied in exploring those questions. Weinberg (1971) is generally credited with first suggesting the approach, though that first contribution is largely anecdotal. More experimental work quickly followed, however, with Sime, Green, and Guest (1973), Weissman (1974), and Lucas and Kaplan (1974), and since then a great deal of exploration has developed. Much, however, has involved immediate and interactive software with particular attention to novice subjects. Schneiderman (1980) surveys the human factors involved in software generally, and sees programming language as the very deep end of such study. Perhaps because of the strong feelings held about the subject of programming languages and programming in general, especially at the height of the "structured programming" debate, there has been informal doubt and scepticism about the psychological approach. Brooks (1980) looks critically at the methodology of experimentation and is concerned about the frequent lack of rigorous application of procedures developed over years of psychological research. But Sheil (1981), in his survey of the approach is very harsh in criticism not so much of the method, but of the conclusions drawn from them. Green, Sime, and Fitter (1981) discuss the difficulties in dealing with the notational aspects of programming language, noting especially the extraordinarily wide scope of psychology involved. Curtis (1983) provides a survey of the more experimentally practical work particular to programming language design, and generally seems optimistic. Curtis focusses, however, on the detail of structure in procedural languages, and does not really consider any wider possibilities or implications. More recently Wickens and Kramer (1985) provide, from the side of psychology, a survey of the application of psychology to engineering in general, particularly including computer software and programming language, and seeing the situation as diverse exploration from several different perspectives, notably human factors engineering and human performance research. In general, it appears that where rigorous technique yields significant results, the depth and breadth of implication are not yet seen sufficent to influence programming language design.

In applying psychological experiment to programming languages, two paths of exploration have proven popular. From one direction has been examination of topical controversy from computer science, most notably experimenting with different structures for sequence control in conventional "navigational" languages. Curiously this interest seems to have been sparked outside computing in interest about decision making (Wright and Reid, 1973, for example), but there is now a surprisingly large, and continuing (see Vessey and Weber, 1986), literature on this one point. The question is frequently related to the more direct implications of the debate on structured programming. In some cases, several topics are considered all at once in sets of language features, such as those of Gannon (1976)'s ambitious study involving several "good programming" controversies in one experiment. From the other direction has been examination of theoretical propositions from cognitive psychology: short term memory and "chunking" of structures together (from

Miller's 1956 classic "The Magic Number Seven Plus Or Minus Two"). In some cases, general psychological models are constructed, such as the network and process model suggested by Carroll and Thomas (1981, 1982), and the "information-processing" analysis for human behaviour proposed for experimentation by Card, Moran, and Newell (1980, 1982).

As an approach to exploring language values, psychological experiment has several advantages. Of initial interest: it is a method that obtains "actual" and theoretically repeatable results - statistical data enabling a variety of comparative analyses. Hypothetical developments may be examined without the immense effort and investment of language development and support, which is a significant attraction. Moreover, as investigations may concern details or general models, it could perhaps lead to inductive inquiry and foster more understanding. And beyond direct reasons is the benefit of the establishment: cognitive psychology, while young, is a discipline with primary aims that clearly encompass those of language quality and evaluation - perhaps it could serve computing as a guide?

While research in the area can claim results, little of the work shows in language design yet established. Perhaps the Ada semicolon is a terminator because of the study of Gannon and Horning (1975), to chose a small example, but perhaps because the influence of PL/I was strong. There are so many factors it is very difficult to be certain of anything. Psychological research is new in application to programming languages and they take some time to become established - perhaps the connection may yet develop.

The approach does have drawbacks, however, some perhaps serious, especially with regard to the particular concern about programming language diversity and longer lived programmes.

Expense & Experience: Programming remains, fortunately or unfortunately, a skill much in demand, and in order to afford many experimental subjects, research commonly examines university students at an introductory level. It is not clear that what is learned about these subjects will apply to more experienced programmers.

Interpretation & Interference: The problem being explored in experiment may often involve general classes of which the experimental apparatus reflects only one instance. It is often not clear that behaviour discovered in this instance is applicable to all members of the class. It may concern the application paradigm: is a language best for business data processing best also for scientific calculation? It may concern the notation itself: do changes in lexical structure affect application of grammatical structure? Sime, Green, and Guest in their classic study of language conditional structures, use particular syntax ("BEGIN, END") to stand for more general (block structure) that is the object of the study. Similar assumptions are made in continuing explorations of the same study (Vessey and Weber, 1986, for example). A particularly disturbing example of this problem occurs in an experiment by Ledgard et al. (1980), a very widely published model for psychology and software research. After simply, though rigorously, showing one "English-based" text editor better than an equivalent "notational" editor, it is concluded that any English based editor is superior to all equivalent notational editors. In fact, very little is shown at all: some "notations" may well be poor, others may not. The conclusions of the experiment are not convincing at all.

Time & Tools: Controlled experiments cannot last very long. It seems possible, however, that many effects of notational tools are not realised until much familiarity has been achieved - it isn't clear how long it takes, or whether controlled experiments can manage that long. Especially in experimenting with application in mind, time itself may be a factor, as short term and long term application techniques might well diverge.

All these problems are ones of interpretation, and in each case the danger is overgeneralisation. This is understandable - we test what we suspect - but when applied to such large and complex situations is far too limiting. And in light of knowledge about how surprisingly adaptable human behaviour is, it is not realistic. Even change itself can influence behaviour, and it is not clear that programming should be exempt from "fashion". Sometimes the problems are recognised, but are considered problems of insufficient resources. The results would have been convincing - if more people could be coopted - if more time could be taken - if more cases could be studied. More is not available in the quantities needed, nor is it likely to be. In order to explore on the scale required, the apparatus must be more complex and diverse, the control must be less rigorous, and the observations must be more difficult. This is not experiment: it is history.

One line of psychological research conducted by Perlman (1984) is interesting in that it does concern itself with the low levels of interplay in programming language use. Perlman draws on a number of sources from natural language research and explore the relationships between symbols, names, and concepts. His experiments appear to show a number of trade-offs occur at the low level, relating, for instance to application of lexical or semantic knowledge. Perlman uses these findings directly in designing systems where the immediacy of interaction makes the choice between tradeoffs clear, suggesting "menu" interfaces for interactive use, and applying the strategy to a command language. For the longer term, though, and at higher levels, the questions are still difficult. "Semantic knowledge" is a much encompassing term. It seems reasonable to expect more tradeoffs, but experimentally exploration will be complex. Perlman explicitly rules out concern of this sort, and so loses some considerable import when experienced and longer term software users are considered. And this is a main interest in practical programming language design.

Most careful work that has been done in applying psychology to software shares this approach. In the design of calculator "key-pad" language, for instance, Mayer and Bayman (1981) are relatively free from concern with long term issues because calculator use is immediate and fleeting. The next step taken is usually to programming languages for people just learning to programme, and here too, Mayer (1981) for instance, can assume the relatively major changes happening in relatively small times will make significant conclusions possible. More than this, however, proves difficult. Trancz (1979), for example, maps out a cognitive processing model for programming language use. But he explains that while there is general agreement on distinction between, say, short-term and long-term memory, the exact structures involved and capacities of both structures and connections are not sufficiently established to rule on detail necessary for relevant distinctions in actual programming practice.

This is the main point Sheil (1981) makes with regard to programming "notation", that the experimental view of the programming (as opposed to more immediate user interaction) task is much too shallow. He shows how, from a psychological view, almost all programming language experiments neglect

possibly quite important factors, the methods used to teach the language, for example, and even then manage to turn experimental results that are quite equivocal into apparently insightful conclusions. Sheil questions whether they might not really show something else inadvertently:

"First, given the small sizes of and inconsistencies among the reported effects, it is not even clear that notation is a major factor in the difficulty of programming. The study of programming languages has been central to computer science for so long that it comes as a shock to realize how little empirical evidence there is for their importance. Second, many of these effects tend to disappear with practice or experience. This raises some doubt as to whether these results reflect stable differences between notations or merely learning effects and other transients that would not be significant factors in actual programming performance."

One factor leading to the small significance often resulting is simply that it is usual to compare schemes where there is doubt anyway - few experiments compare APL and binary machine code.

While there is much enthusiasm for the application of psychology to computer science, the questions about programming, and then about programming language, seem to be the most difficult. Wickens and Kramer, theirs one of few surveys external to computing that do encompass the programming efforts, merely record views and claims one way and the other, and concentrate on the more immediate human-machine interactions that do not so easily defy analysis.

And perhaps surprisingly, no other notation more limited than natural language or more complex than direct interaction seems to have been singled out for detailed psychological study. While this means psychology attention is attracted to a computing problem, it also means there is little applicable experience. The application of psychology to programming language design is fascinating but not conclusive, perhaps not even convincing - understanding the interplay of thought and notation simply requires greater understanding. Method there is; theory there is; authority there is not - not yet, anyway. Ergonomics is a proven design approach, but is founded on and limited by understanding of the human factor. Where the human factors involved are well understood, the importance and success of ergonomics cannot be doubted, for example in the classic telephone handset design of Dreyfuss (1955) based on hand and head physical proportions. But the proportions inside the hand and head that bear on programming language design are simply too complex to be well enough understood (yet). It is quite reasonable and perhaps important that the psychology of computing be further explored. The approach is far from silly, but it is far from definitive. At this stage it is not psychology that can serve computing as a guide, but computing that can serve psychology as a subject.

Computing as a subject of psychology might be seen as an approach to the topics grouped as Artificial Intelligence. And this area of computing has certainly been the fount of much interest and influence in programming language design. Certainly Lisp has been a "test-bed" for some time, and recently much attention has been focussed on Prolog (see Cohen, 1985). But the interest and influence is not unique, almost every programming paradigm seems able to contribute to language design in general. It does sometimes seem suggested that because Artificial Intelligence is concerned with some human behaviour distinguished as "intelligent", that programming languages designed to model such behaviour should possess some superior quality. Winograd appears to hold this in imagining beyond programming language (1979), and it appears even more so in some appeals for "object-oriented" programming environments such as

Ingalls, 1981, or Krasner, 1983, for example). Even assuming the premises of the argument, however, it is not clear why a notation designed to model one context need be the best to model another. Just as it's not clear that the most "intelligent" people can perform all tasks better than other people, so it's not clear that a language best enabling "artificial intelligence" is best for less glamorous applications.

Empirical Analysis

The importance and elusiveness of the human factor in understanding language application does make the idea of data - any empirical data - very enticing. Largely because of this appeal, one approach to language exploration involves the study of the empirically tractable artefacts of the programming process: usually the programmes themselves.

Knuth (1971) was an early proponent, and claims to have begun by studying waste punch cards - though later moved on to study programme libraries. Knuth's main interest was in language implementation, and he sought to determine from existing programmes the most frequent, statically and dynamically, language patterns in actual use. Many analyses have followed of many different programming languages, both statically and dynamically, mostly concerned with implementation implications. Halstead (1977) too saw empirical analysis as important, and this is the practical basis of his "software science". His interest was also not programming language, but programmes themselves, seeking to characterise various elusive textual properties of programmes. However, the "science" is not really completely accepted. Though regarded as very promising in some reviews (see Fitzsimmons and Love, 1978, for example), in others severe doubt has been expressed about both theoretical basis (Coulter, 1983), and empirical support (Shen, Conte, and Dunsmore, 1983). While the methods seem to have some attachment to practice, it seems difficult to say just what it is, and how much may then be reasoned on that basis. However, such measures are still used, and indeed wider applications are even suggested to mathematical and scientific notation generally (Patrick, 1983).

Reasoning from empirical analysis is inductively founded, examining past conditions for past outcome, then considering future conditions for future outcome. It does permit large collections of observation, and by its retrospective nature can appear to avoid the need for rigour so necessary in experiment. Results in the area are seldom strong or very surprising, but may well remind about the imperatives of practice. Knuth, for example, showed that many assignment statements in Fortran are simple increments. Sale (1981), on the other hand, claims that Pascal programmes very rarely contain the "repeat" statement. Following such studies is generally consideration about current practice. In the first case, for example, we might consider language refinement: if increment is so common it should perhaps be given special treatment. In the second case we might consider language reduction: if "repeat" is seldom used it might be modelled modelled with "while". In the same way, for example, Coulter and Kelly (1986) report a study showing assembler programmers little use many instructions, and suggest that impact of reduced instruction sets might have less impact on low level programming than might be supposed.

On a more day to day basis, empirical analysis has been suggested as a means of examining particular programmes. One application discussed has been in evaluation of student's programmes in and education and assessment situation as proposed by Robinson and Torsun (1977). (See Rees, 1982, for a Pascal context, or more recently the detailed discussion of Redish and Smyth, 1986, using Fortran 77). Such methods make all sorts of presumptions, of course, and in the assessment role pose some difficult questions. In a recent study, for example, Gannon, Katz, and Basili (1986) empirically examine programmes written by newcomers to Ada. It turned out that the programmers failed to use Ada in the way the experimenters regarded as most desirable, even though they were experienced programmers, trained in Ada, and knowledgeable about the application. So it was decided that even knowledgeable and experienced programmers need special training to use Ada properly. Of course, this may well be true. It may also reveal something about Ada, but that was not considered.

However, similar methods of analysis can also be used simply as another way to exploring programmes, and in so doing, in a background way of course, exploring programming languages. In this way too various programme driven software may also be useful, profilers, structure diagrammers, even formatters - the ubiquitous "prettyprinter". Similar software is also available for natural language documents, spelling checkers, document formatters, and also more sophisticated programmes for concordances (such as the portable Oxford Concordance Program by Hockey and Marriott, 1980), and diction and grammar analysis (such as the Unix Style and Diction programmes by Cherry and Vesterman, 1980). In fact many of the applications of computing to natural language literature (see for example Sedelon's 1970 survey, or Hockey's 1980 guide) could well be considered for programming language "literature" too. One interesting example of this combination was by Morgan (1970), who used spelling error detection methods integrated with a programming language to detect probable semantic errors. As with the tools commonly used with natural language, such programming language tools might not be able to prescribe, only to suggest, but that has proven useful.

Programmes have been used as empirical data in the study of language, language implementation, and student programmes, but programmes are not the only artefact of the imperfect process of programming. "Programmes" often include errors, and error reports, as artefacts too, can also be recorded and studied. This technique has been considered primarily to study error diagnosis, as suggested by Wetherell (1977), a challenge picked up by Ripley and Druseikis (1978). A main interest here was in the possibilities for automatic error correction, like that done in the PL/I variant PL/C (Sprowls, 1972). But it also lead to rough knowledge usable in simpler application. The approach seems to have the potential to tell something about language application in general, if not as a direct report on language design, then perhaps as an aid in programming instruction (see, for example, Biddle, 1984). But as there concluded, it is very difficult to reconstruct the causes of errors at that distance, and the approach seems useful more as an aid in indicating the need for attention. Otherwise the direct empirical approach of Pugh and Simpson (1979) in actually personally examining errors as they occur in the context they occur seems bound to be superior, though tedious - perhaps language implementations could design error reporting explicitly to aid this.

Programmes and error reports are attractive in language evaluation because they are empirical: they are doubly so because they are machine readable. It is relatively easy to "record" data from many sources and

study it later - it might also be possible to construct "nosy" programming language implementation software to record or even analyse data habitually. If analysis over large practice is too abstract and remote, coverage could be reduced or otherwise structured so to bear on some particular application. In fact, the approach might also be useful in programme language implementation if by extending language software "memory" structures to include not only "input" but also "output" so enabling some bypass. In this way language software could "remember" input patterns and implied implementation without the need for detailed re-generation. The approach might be especially useful in iterative and cumulative development, and might be seen as specialised extension of "probabilistic" language (see Wetherell, 1980), or of "incremental" language implementation systems (see Ghezzi and Mandrioli, 1979), usually associated with language syntax oriented editors and environments. (See Medina-Mora, 1981, or Teitelbaum and Reps, 1981, and Reps, Teitelbaum, and Demers, 1983.)

It does seem an easy approach to understanding language use: simply watch what actually happens (on a large scale), and hope to learn. It is clear, though, that the past is not an easy way to see the future: empirical analysis is not clairvoyance, it is more like archaeology. There may indeed be lessons to be learned, but whether they are lessons that apply to us - or are of interest to us - is not easy to discover. Presence or absence of simple lexical structures is easy to detect, but their interrelations - and many many interrelations might be of interest - are much more difficult to sort out. Even when hypotheses can indeed be formed, it can often be difficult to really determine cause. At this distance from the programming activity, motivation and causality are unclear. Moreover, should the sought be found, it might not prove conclusive. Empirically the popular idiom will dominate, and the connection between popularity and quality must then be

There are many programmes in the world, and many that last but briefly. This approach is easy in ways others are not - large scale observation can be automatic. It is all too easy, however, to systematically collect great volumes of data only to glean little knowledge and less insight. Empirical analysis does lead to illumination, and may lead to resolution. But it is not an easy procedure, and in application to programming language has not hinted of resolution to convergence.

SECTION III. CONTEXT

As seen already, programming language design does often directly or indirectly use as precedent some linguistic context before or beyond computer programming. It is the human factor that hinders understanding of programming language quality and convergence, and many of the resulting problems are similar to problems in all notation - indeed all language. Accordingly it seems indicated to at least explicitly inspect the these problems of quality and convergence in the linguistic contexts that are precedents.

It's often said that programming languages are "artificial" languages, rather than "natural" languages, but this distinction is not clear. It is people that change, develop, and control English, say; and even specifically created languages such as Esperanto soon have native speakers. Computing itself is not new, and the more general task - algorithm - and the more specific task - programme - are both ancient, as Knuth shows in his exposition of "Ancient Babylonian Algorithms" (1972). In advocating language programme "synthesisers" rather than general text editors it is often claimed that programmes are, after all, not text (Teitelbaum and Reps, 1981). But what is text? The difference is only that programming language text is more rigid in structure: strength and a weakness both, as all who learn programming discover. Especially with regard to computing machinery, our modern development has concerned precision, but only precision on one side, on the other - human - side, precision is not a clear concept. The distinguishing characteristics of "programming language" are those stemming from operational use: human specification of automatic computing for various applications where automatic computing seems desirable. That's all.

The notation often thought most akin to programming language is that of mathematics. Mathematical notation as a whole is truly vast and complex, but the study of Cajori (1928) does yield wide insight into the historical processes that have governed notational evolution. Regarding the formal definition beneath notation, this is a history of growing capability. Regarding the human factors of that notation itself, this is a history of politics, society, and accident. There do seem to have been success and failure in notation, but the relation between the mathematics and the notation seems unclear in function and even causality. Arithmetic may be preferred in Hindi-Arabic notation rather than Roman; calculus may be preferred in the notation of Leibniz rather than Newton. But it is not clear mathematics advanced because of either, and it is not clear mathematics will advance further because of either. There are parallel questions in programming language application, but indeed there are parallel questions in any language application.

What are the ties between programming and more general natural language, and what does study of the latter suggest in study of the former?

From Programming Language to Natural Language

In relating programming language and natural language is, firstly, the idea of linguistic precedence clearly followed in programming language design. Whereas earlier we discussed the possibility that more closely following natural language was a clear direction for programming language, the claim here is that, to

a certain extent, it has followed it anyway. Or rather, that the two are related sufficiently that knowledge about the natural case pertains to the case of programming.

Programming language has been necessarily written, and so is particularly influenced by written natural language, itself largely secondary to and derivative from spoken natural language. Much programming language design directly follows natural language, as is particularly clear with, say, Cobol (see Sammet's retrospective, 1978b). Much that follows some other notation indirectly follows natural language, as is clear with Fortran and numerical mathematics (Backus, 1978b), the mathematics itself largely influenced by a variety of natural language usage (Latin "et" to "+", for example - see Cajori).

Even in more recent, international, and "general purpose" languages such as Pascal there are English words and abbreviations of words. Most programming languages extend the mathematics practice of using natural language graphemes, letters, as names and allow sequences of letters. This is an invitation to allow "words" - or word-like sequences - from natural language: a practice often pedagogically encouraged. And the general graphical structure of symbols oftens model natural language. Not only are familiar letter symbols used and strung together as words: linear structure is left to right along lines and the right end of one line is followed by the left of the next line - even punctuation is similar.

And obviously there is no question about which particular natural language family, natural language, and usage is most commonly followed. In context, origin, and power there is a concentric focus: European, English, and American. And so a programming language might employ a grammatical structure familiar in one language, yet unusual or unknown in another. Statement structure represented "infix" might be familiar in English, yet in other, even other European, contexts, "postfix" might be more familiar. And even while incorporating some changes in adapting to another "naturally" linguistic orientation, programming in other contexts frequently do retain, in integration, those contexts with existing programming language design. So within the contexts of several different natural languages there are programming languages with appropriate keywords, and programmes with appropriate names. There remains the orientation of page structure, discrete graphemes, and alphabet, and many natural languages not compatible are adapted. As indeed was English once adapted to the foreign written form it uses today(!).

So it is clear the natural language backdrop does frame programming language and not always with the most "natural" of natural languages. And in consideration of that relationship it would there are also seem that there would be important secondary parallels that might yield insight in programming language matters. But at this point, if not before, there must be concern about the lack of knowledge about the nature of natural language. Not only is there little that can be done because of this, but that observation is itself wound up in the larger problem faced.

What is the function of natural language? Already there is difficulty: it is tempting to answer "communication", but even that leads to great philosophical tangle. The function of natural and of programming language seem sometimes very similar: merely provision for manifestation. In fact, everything that might be done with programming language might be done with natural language. It might be argued that this is always the case anyway, as no one is born speaking Algol. What is almost always done, most importantly done, with programming language is specification of computing in some context where automatic

computing might be useful. But even with such specification the interplay, and it is vital interplay, of expression and comprehension in community is done generally with natural language.

It seems an important facet of natural language that it is flexible, that it can adapt in new circumstances and "survive" through evolution. Moreover, such change is organic and determined through usage in the natural language community. In looking at programming languages, this might be considered in two ways. In a direct way, programming languages are explicitly made and can be explicitly re-made, or rather new ones made. The communities that created programming languages can abandon them, adapt them, and create others. Formally, the computability scope remains. Evolution might proceed less smoothly than with natural language, but proceed it does. Indeed, that is how the diversity problem arises. In an indirect way, language use may evolve as language users do. As individuals and communities simply use a programming language, as they use a natural language, so the associations of such use become established: so may they cease, change, and adapt. Programming language evolution in this way parallels natural language evolution well.

Dijkstra (1981):

"The ACM has a Special Interest Group on Programming Languages, but not one on programming as such; its newest periodical is on Programming Languages and Systems and not on programming as such. The computing community has an almost morbid fixation on program notation, it is subdivided in as many subcultures as we have more or less accepted programming languages, subcultures, none of which clearly distinguishes between genuine problems and problems only generated by the programming language it has adopted (sometimes almost as a faith). For this morbid fixation I can only offer one explanation: we fully realize that in our work, more than perhaps anywhere else, appropriate notational conventions are crucial, but also: we suffer more than anyone else from the misunderstanding of their proper role.

The problem is, of course, quite general: each experienced mathematician knows that achievements depend critically on the availability of suitable notations. Naively one would therefore expect that the design of suitable notations would be a central topic of mathematical methodology. Amazingly enough, the traditional mathematical world hardly addresses the topic. The clumsy notational conventions adhered to in many mathematical publications leave room for only one conclusion: mathematicians are not even taught how to select a suitable notation from the established ones, let alone that they are taught how to design a new one when needed.

This amazing silence of the mathematicians on what is at the heart of their trade suggests that it is a very tough job to say something sensible about it. (Otherwise they would have done it.) And I can explain it only by the fact that the traditional mathematician has always the escape of natural language with which he can glue his formulae together.

Here, the computing scientist is in a unique position, for a program text is, by definition, for one hundred percent a formal text. As a result we just cannot afford not to understand why a notational convention is appropriate or not. Such is our predicament."

The application of programming language especially in automatic computing has lead to requirements of formality and precision. But the facets of human application of programming language are not formally tractable, nor even well understood. In actual day-to-day practice, moreover, natural language is, in legal or military applications for example, sufficiently formal to determine life and death. In the context of formally defined computing, the implementation of the programming language , Dijkstra is quite right: a programme is "one hundred percent a formal text". But regardless of formally defined computing, there is no human meaning without human context. In the context of human factors in computing, the application of the programming language, Dijkstra is quite wrong: if there is sense in which a programme is a totally formal text, it is one not understood.

In suggesting these ties between programming and "natural" language, there is the question of question of proof: how could we know the phenomena were one? Without understanding the nature of

language itself, it is difficult to see how the question might be determined. Assuming a neurological basis for all human behaviour, a neurological view might prove helpful. Knowledge relating higher level behaviour to lower level neurological phenomena is very limited, but an approach potentially indicative might consider abnormal, rather than normal, human behaviour. Particularly, it might be investigated whether people who seem to have natural language abnormality also seem to have programming language abnormality. Most significant might be such a relation in cases where abnormality is directly physical rather than possibly social. "Aphasia" is the very general name for such language abnormality, it is usually a consequence of severe physical brain damage. Aphasia research has shown remarkable relationships in language and language-like phenomena, but perhaps posing more questions than offering answers (see Gardner, 1976 and Gazzaniga and LeDoux, 1978). Clearly this offers a practical route to verifying speculation on the biological "implementation" of language, but not (yet) to the degree where reverse engineering of language design has been undertaken. If stronger evidence is desired to link programming and natural language, research in this direction seems worth consideration - none appears to have been undertaken.

From Natural Language

In considering programming language diversity, the difficulties in reasoning to convergence involve the programming language facets that parallel natural language. There is vast experience with natural language, so the parallel might prove helpfully illuminating.

Natural language is a central concern in many many disciplines. Linguistics itself, but also then wider and wider interests in humanity: semiotics, psychology, sociology, anthropology, and philosophy. And quite practically: translation, lexicography, and literature. In considering reasoning about programming language, the problem is one of quality and context: how to reason about quality? In considering reasoning about natural language, the problem is the same. The literature is immense. And the problem is deep: it's no longer of question of supposing that any particular language is more "natural" - as was once claimed of Latin for example (see Gleitman, 1986). At the root of the problem lies a fundamental dispute about the nature and basis of language: the question of language "universality" or "relativity". To greatly abridge the two views: If universal, all language shares a common "deep structure" of knowledge and thought. If relative, any language is a world unto itself, a thinking kit wherein distinction between language and knowledge is blurry. The distinction is critical: without a universal context for language, there can be no reasoning in confidence, to comparison or to convergence.

Universality is central to the linguistics of Chomsky in particular (see Chomsky, 1965, and 1968) in his search for a minimally sufficient mathematical model capable of encompassing natural language. However, there is the long tradition of other linguistic knowledge and the background of "the Ghost in the Machine" that Ryle (1949) attributes to DesCartes. It's also the basis for most artificial intelligence work in natural language understanding, computational linguistics, and automatic translation. Certainly there are well known examples that might seem to support the view: perhaps the best known is that made popular by Papanek, 1971 (but after Koehler and in fact after Usnadze).

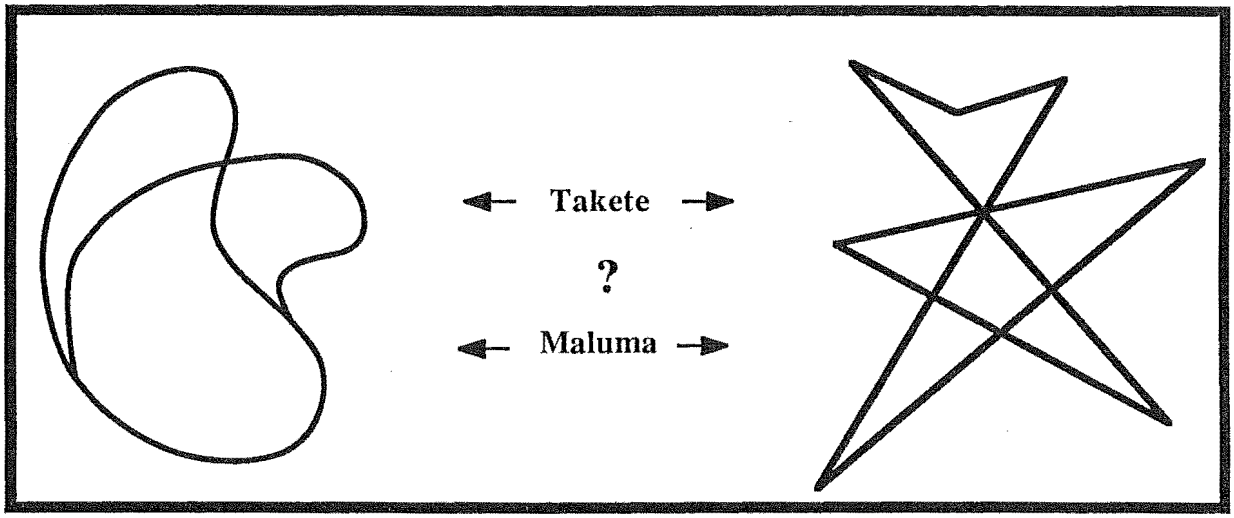

Figure 2.3.1: An example of macthing words and images - the suggestion here is that "Maluma" matches the image on the left, and "Takete" the image on the right. While apparantly this is the matching popularly made, it is not clear proof of universal components in all language. [After Papanek (1971), itself after Gestalt theories of Koehler and Usnadze]

The example shows the two figures and two "words", and invites matchings of figures and words. And in practice by far most people match the rounded figure with "Maluma", and the jagged with "Takete". There are several difficulties with such an example. Even the claim made that the "words" are not present in any known language is open to doubt, as parts of the "words" are suggestive of words in, say, English and Maori. But there is also the question of the importance of language realisation. For instance, the distinction may be that of speaking the words, and may be lost or reduced in reading or writing them.

More recently, the celebrated study of Berlin and Kay presents an effort across natural language and across culture to find an example of a natural language universal (1969). In this very rigorous study, natural languages were examined for terms for colours, and it was discovered that there was evidence of a structure of distinction across languages. For example, while all languages had terms corresponding to black and white, in the cases where there was a third distinguished, it was always red, and if a fourth, it was either green or yellow, and so on. The problem here is that as linguistics research this is concerned with description, and there is interplay between the language and the culture, and the culture and the environment. So it seems reasonable to imagine that the discovery is not only about language, but about the world. In linguistics this is now accepted as part of the descriptive orientation of the subject. In computing, however, the concern for prescriptive orientation is essential: it is actively engaged in language creation. So while the colour study may suggest detectable universality in the world of natural language, it's less clear what it says languages that have been specifically designed anew. The search for insight and proof in the matter is very widespread, and many aspects of language or language use have been examined for suggestions of some universal basis to natural language. There's great variety: Bickerton (1973) even suggests that creoles, informal language combinations, illustrate the common roots of all natural language. No study appears to be helpful in programming language design: it is difficult in evaluating to separate what is to be language from what is to be the world. Are languages necessarily vocal? written? linear? Of course the questions are

fascinating and invite continuing exploration. But even should some universality be established there is no assurance of finding some ideal Rosetta Stone enabling ideal translation.

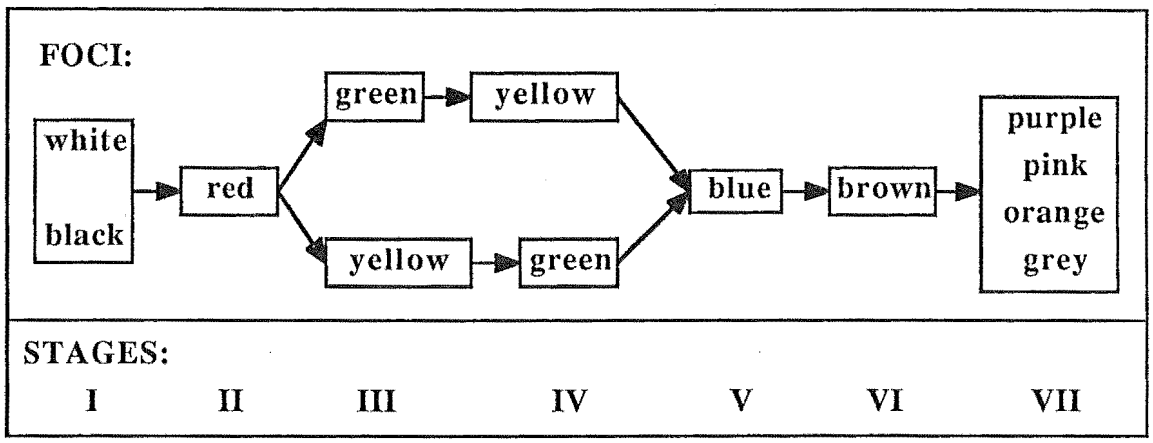

Figure 2.3.2: Apparant hierarchy in colour terms across languages - Showing what colour foci are distinguished in language, across stages of languages that distinguish numbers of colours.This structure is suggested as possible proof of universal principles across languages.
[After Berlin and Key (1969)]

Language relativity is central to much modern philosophy, especially as many, for example Ayer, would say it is indeed an inherent concern of all philosophy (1936, now 1971, and more widely in his review of twentieth century philosophy, 1982). Ayer: "The propositions of philosophy are not factual, but linguistic in character - that is, they do not describe the behaviour of physical, or even mental, objects; they express definitions, or the formal consequences of definitions." The relativity of language and its implications are certainly central to both (first against and then for) Wittgenstein's philosophies - "the limits of my language are the limits of my world" - (1961, 1964), and also Ryle's (1949) critique of Cartesian dualism, Quine's concern about the indeterminacy of translation (1953 and 1960), Austin's claims for legitimacy for "speech acts" (1962), and Goodman's exposition of the flexibility and subtle implications of language frames (1968 and 1978). And of course there are a myriad roots and branches. In more accessible concentration on communication, McLuhan (1964) too relies on the relativity between "medium" and "message". And this is also the context of criticism of some directions in artificial intelligence involving natural language understanding, for example Robinson (1975), and at least one aspect of the criticisms of Dreyfus (1972) and Weizenbaum (1976). In distinguishing "computer power" from "human reason", an important point Weizenbaum makes is how central a difficulty in conceiving "artificial intelligence" is the encompassing nature of the human context. This in itself need not be an indictment, as Smith (1980) or Pateman (1981) argue in discussing some essential philosophical advantages of human communication with non-human human-modelled "intelligence": intelligence is not all that is human. However, in the context of programming language design the facets of humanity involved may well be significant, and the more abstract philosophy of (possibly) artificial intelligence - such as that outlined by Boden (1977) or Sloman (1978) for example - may well be of limited application. For human use, human feelings and behaviour are important regardless of "intelligence".

The matter of linguistic relativity has involved practical discovery or verification, most notably the work of Sapir (1931) and Whorf (1956) with regard to to cognitive anthropology is strongly associated with the view:

"The background linguistic system (in other words, the grammar) of each language is not merely a reproducing instrument for voicing ideas but rather is itself the shaper of ideas, the program and guide for the individual's mental activity, for his synthesis of his mental stock in trade. Formulation of ideas is not an independent process, strictly rational in the old sense, but is part of a particular grammar and differs, from slightly to greatly, as between different grammars."

The view may seem shocking at first, staggering in implication. Perhaps the staunchest objection might be the prospect of solipsism and nihilism that seem consequent when the recursive involvement of language in studying language is realised. There is the possibility that both relative and absolute grounding are together involved in the answer. There is almost unquestionably some relative factor in the self-reference of natural language, though it may be unpredictable. Name and meaning are related, perhaps as book and cover. It's popular to bind language and instrument together in metaphor: Iverson's (1980) theme: "notation as a tool of thought" is a good example. But the whole metaphor can be misleading: the human-instrument relationship is never merely one way. Any tool affects the way in which the task is thought about.

But there is also almost unquestionably some absolute factor in the community aspect of natural language. Any new born human child can learn any human natural language, but no member of any other species can - even though this has been reasonably closely examined with chimpanzees, for example (see Limber, 1977).

Even were the outward command of language no barrier between species, there would remain the gulf of "culture". Wittgenstein (1953) remarked that "if a lion could talk we could not understand him", and Weizenbaum (1976) suggests there would be a similar problem with artificial intelligence. In practice humans and animals can and do intercommunicate to some extent, even if only eavesdropping as with bees, for example (Von Frisch, 1967), so perhaps even this scale of enquiry is too narrow. Perhaps there is an evolutionary continuum of community. Perhaps there is a universal structure underlying language, but at such a low level reasoning with that universality is difficult. Perhaps there is, even formally, the possibility that the question is beyond the unresolved, unresolvable. Even should human behaviour be known equivalent to a computing model, even should an equivalence mapping be known, even should the mapping be operationally possible, even then human behaviour might not be understood.

In spite of the difficulties in understanding how natural language works, quality of natural language is still informally sought and recognised. Poetry exists - but it is not language engineering. And in mundane practical matters, lexicography and grammar is still useful - but it has been accepted for some time that these are descriptive, not prescriptive, disciplines. This philosophy is often regarded new and radical, probably because of the social and cultural use of language and language distinction (see Halliday, 1978, for examples). But the philosophy is attributed to Franz-Passow in 1812, and has been dominant in English lexicography since the work of Murray in the design of the New (Oxford) English Dictionary (Murray, 1979). Lightfoot (1982) gives an overview of a current view of the tasks of practical linguistics as a descriptive and inductive science. The relativist view extends to the fingertips of natural language, and even orthography and typography have a role to play.

Programme presentation involving these concerns is regarded significant by Bishop (1979) and Sale (1979ab), and one of the motivations of Knuth in the design of Tex (1979). And the same kind of concerns apply not only in language but in painting, in sculpture, in music, in dance, all the parallels drawn by Goodman in his analysis of "Languages of Art". In all such fields there are many traditions but little or no formal rules. Every facet of humanity may be involved: logic and emotion - change itself may even play a role. Great studies in art have centred on just such extra-logical influences, Kandinsky's (1912) examination of spirituality in painting, for example, or Graves' (1925) thesis on irrationality in poetry. In addition, it must be acknowledged that a major wide theme of art in the twentieth century has been the determined exploration of both new media and structure, even beyond the limits of taxonomy. For example, consider the work of McCahon involving written words integrated with paintings - simply directly "saying" what seems best said (see Brown, 1984). Much work done by artists since, say, Matisse, has focussed directly on the issue of representation, and has shown that there is a very wide scope for efficacy (see Rosenburg, 1975, for example). Each approach offers leads to its own constellation of feelings, and questions of form and function are lost in confusion between palette and canvas. Reasoning from this interplay, Arheim (1969) develops a relativistic psychology of art, wherein he can only conclude that too often the difference between "seeing" and "thinking" is simply false distinction. In more recent art the growing front of "Postmodernism" goes even further, exhibiting irreverence in exploration of representation that explicitly toys with convention and self-reference to the medium itself (see Wallis, 1984, for examples), form often mocking function. And such re-thinking is not limited to the "abstract" fine arts, but indeed grew up as well in such practical pursuits as graphic design and architecture (see Partoghesi, 1983). Programming language design doesn't seem such a different area: perhaps there will be "postmodern" programming? Just as there are many application areas of programming language now that were ignored in the past, so might the future encompass even more variety.

Probably the best justification of this approach with regard to more than the arts is Feyerabend's "Against Method" (1978). While even Popper (1959) incorporates the need for the exploratory nature of science being more than deduction, Feyerabend's claims are rather stronger, focussing on the danger of the working to pattern - what Kuhn (1962), for instance, claims to be the great strength of the scientific approach. But while it's interesting to note that there is not agreement on the subject even in the strictest science, programming language design is not that strict - the human side of that design seems clearly within the humanities. There is no known context of contexts, no basis for reason with confidence, to comparison or to convergence. There is no reason not to suppose the same mix of relative and absolute in the human application of programming language.

Generally the problems of relativity in programming language quality are little discussed. When they are recognised, they are almost never dwelt upon. Curtis (1983), for example, about to survey psychological work regarding language characteristics: "The Whorfian hypothesis suggests that people's ability to think is limited by the language they are thinking in." But leaves the issue there. And the discussion is often concerned only with the question of limitation of performance. Some extensive texts on programming languages touch on the problem directly. MacLennan (1983) speaks of the ability of a language to "facilitate or impede". Ledgard and Marcotty (1981) do state that the hypothesis is that a language may

have the ability to limit what "one can think". All this underlines the severity of the problem. No one discusses how a language might lead one to use recursion instead of iteration, use a different algorithm, invent a new algorithm, think of a solution to a similar problem, think of a whole new approach to the area, or give up altogether.

But there are cases that have been of consequence in science, many involving visual aspects. Miller's recent (1986) study of "Imagery in Scientific Thought", for example, details the importance - both helpful and hindering - of visual images in the development of sub-atomic physics. The importance of the greater possibilities of more free graphical representation is now also being recognised in some programming language design. Chang (1987) surveys current visual languages, and finds that not only figurative, but also schematic or "iconic" visualisations at least appear to be a useful way of programming for some applications. Sloman (1985) attempts to explain the perceived importance of such varied representation in cognitive terms, arguing that it is a mistake to imagine knowledge-based artificial intelligence systems on the one representational schema, first-order predicate logic, advocated by Kowalski (in an earlier debate on the subject, 1980). He claims that experience suggests that much knowledge is learned by analogy, and points out that analogy can be integral with notational representation, as with diagrams. Of course, the importance of this argument is reduced if the components of analogy can themselves be represented by logic. But the disadvantages there are the same disadvantages of any layered implementation: comparative inefficiency, and the potential for hindering structure. Hofstadter (1981) illustrates some extremes of this case - "Tortoise: ...why don't you do as I do - just hang the record up on your wall and enjoy its beauty all at once, instead of small pieces doled out over a period of time?"

Perlman, in his study of "artificial", by which he means "limited domain" languages (1984) has dealt with the problem somewhat more directly than common, and feels able to be explicit: "... I show that there is no such thing as an ideal natural artificial language, only tradeoffs in design. I show in a series of experiments that the more that is known about the domain of application of an artificial language, the more finely that language can be tuned to the domain, but at the expense of applicability to novel purposes." For all the same reasons, Kent (1978) comes to similar conclusions about data, pointing out that the way in which one structures is related to the way in which one sees reality, and that different views do exist.

So what of the view espoused by Parnas, for example, "On The Criteria To Be Used In Decomposing Systems Into Modules" (1972), that the structure of the data a programme deals with is a guide to the structure of the programme? Surely it must be more limited than first appears, because the subjects of analysis can, in essence, fight back. But it is common to forget this. McKeeman (1975) discusses how programming languages can "interfere" with programming, and urges less use of programming languages, and greater use of traditional mathematics. But he ignores the possibility that mathematical representations have the same "interference" power. The problem applies to all systems of notation, ranging from the commercial orientation of Jackson (1975) in his programming methodology, to the mathematical orientation of Gries: "make the program structure fit the problem structure" (1980). But if the domain is relative, even deciding on the problem may not be easy. In "The Science Of Programming"(1981), an exposition of the technique pioneered by Hoare and especially Dijkstra for formal derivation of programmes, Gries suggests that in the approach he outlines "one programs into a language, not in it". This may be an impossibility.

Most of the severe difficulties discussed suggest a common problem, that of human reasoning about human reasoning. Zemanek (1972) seems to have been the first to discuss the relevance of this very wide subject to computing, concentrating on the philosophical aspect, Davis and Hersh (1980) outline the history of the mathematical situation involved, and Hofstadter (1980) specifically considers the question of computing. Generally the discussion is quiet and shadowed, almost as if it were an embarrassing to have to admit within the purest of sciences such a human weakness. In computing, however, this is especially unfortunate: scientific or not, design and application of new notation is a major practical pursuit. There are two consolations. Firstly, science is not the only paradigm for useful work. Secondly, there may be no alternative. Hofstadter:

"All the limitative theorems of metamathematics and our theory of computation suggest that once the ability to represent your own structure has reached a certain critical point, that is the kiss of death: it guarantees that you can never represent yourself totally. Goedel's Incompleteness Theorem, Church's Undecidability Theorem, Turing's Halting Theorem, Tarski's Truth Theorem - all have the flavor of some ancient fairy tale which warns you that 'To seek self-knowledge is to embark on a journey which ... will always be incomplete, cannot be charted on any map, will never halt, cannot be described.' But do the limitative Theorems have any bearing on people?"

We are left with the informal, the non-convergent, but it is entirely consistent with criticism and theory in design in art and literature. Kernighan and Plauger (1974) in model of Strunk and White (1935), began to discuss programming in a context more inductive than deductive and so honestly face the difficulty. Knuth now binds programming and literacy together in "Web" (1984), attaching importance to the entire presentation of a programme and providing for integration of presentation and programme together. While he claims to have once changed his mind about the original title "The Art Of Computer Programming" (1968 etc.), preferring "The Analysis of Algorithms", he now claims to have changed his mind back (1974a, and Takara, 1985). Like Kernighan and Plauger, Knuth now considers programming "style" as vital, and discusses programmes and programming language as human communication, "literature", subject to artistic consideration. In this way, there is acknowledgement that there are important aspects of programming and programming languages that fit more in the informal context of the arts than in the formal context of mathematics. Floyd (1979) looks at the different ways of programming and programming language that have so far developed and attempts to distinguish "paradigms". And in consciously using the terminology of Kuhn (1962) he also suggests Kuhn's philosophy of the role of the paradigm, not a final, nor unique, not provably correct model for exploration, but a working supposition to be rightly abandoned when as a more useful one arises.

This outlook does not seem to be shared by all. Most notably Dijkstra remains firm in claiming a mathematical view. It can be difficult to understand. About the "correctness" of programmes, he is convincing, arguing that it be regarded as a mathematical concern (see 1981). But he does not seem to admit, directly, that there is anything important about a programme other than the correctness, and he does not seem to consider that correctness of programmes might be regarded as mathematical and yet other aspects of programmes regarded something else. In discussion of programmes and programming he scorns such terms as "ease", "understandability", and "naturalness", and instead advises the sharp distinction between the "convenient" and the "conventional" (1983). Presumedly this latter implies one should choose instruments for their merit in a particular application rather than for their use in other applications. But there can be similar

intent with the abhorred diction as well. In "The Humble Programmer" (1972), he relates how in an informal experiment, he found that the control structures of conventional programming languages "blinded" the subjects from the best solution to a programming problem. Even disregarding questions about experimental technique and the assumption of criteria for a "best" solution: there is no proof, no mathematical proof, that his suggested notation is superior - never mind optimal. Perhaps there is a difference in taxonomy at the root of this difficulty in distinction: in mathematics too one needs be concerned about communication, and perhaps mathematics might be regarded a universal context. If this is the only source of disagreement, any conclusions might yet be in agreement.

As questioning goes deeper, and as the subject becomes wider, it becomes very difficult to rely on firm ground. For a more general example, in consideration of the efforts of the Bourbaki group in attempting to establish a unified structure for all mathematics, Beth and Piaget (1966) argue that the psychological structure of humankind must be seen as involved. And no doubt others would argue that the anthropological ones would too, in accordance with, say, the claims for cross cultural structures made by Levis-Strauss (1963). And so on. And it should also be added that even in reference to fine art, there are those who do seek a unifying deep structure, and who do claim it may be perceived (see Burnham, 1973, for example). And there is work done in semiotics that does lead to understanding of how symbols work, Jung's classic work (1964) of course, but also even work on "paralanguage" of picture symbols made from typographical components in electronic mail (see Asteroff, 1987)!

But while such work is fascinating it does not narrow design potential appreciably at all - it illustrates some paths, denies few, and suggests more. Without deduction to follow in programming language design, there open wide vistas. Ergonomics is sometimes seen a model for programming language design, and specifically "cognitive ergonomics" is suggested in the survey of Curtis (1983). Not only does classical ergonomics involve human factors that are so much more available for examination, but ergonomics is only one approach to design. The relationship between form and function may not be casual, but symbiotic.

In programming language, as natural language, there is the same kind of uncertainty, there is the same kind of inability. Diversity is the consequence, in programming as in nature.

SECTION IV. CONCLUSION

Much has indeed been learned about programming language. In implementation, formal language theory has lead to growing agreement on efficient methods of parsing, and hence to some degree in formal programming language structure. In application, however, while the importance of the human factor remains clear, how best to serve that factor remains difficult to determine. Human factors demand consideration of many special cases in language design: for certain applications paradigms, for example, and for certain people. In any sufficiently accepted context, comparison and even convergence become possible; beyond or between programming language contexts, diversity may flourish.

A recent trend in application software for small computers has been the "integrated system" of several programmes often used together. For instance, probably the first such system was Lotus 1-2-3 (Williams, 1982) and combined spreadsheet, graphics, and database functions. Heering and Klint (1985) have recently suggested that programming languages should go in a similar direction. They argue that programmings currently have to be "polyglots" and must cope with a "hodgepodge" of languages at several levels, and contend that programming would be much easier if all could be integrated into one language. This would be quite true but for the difficulty in deciding what such a language should be. Of course, integration must cost some flexibility, and for some environments the cost is not worthwhile. Probably the attraction of small computer integrated systems is the ease of learning and working with one piece of software, and the cost is restriction to that particular software alone. In the application area, small business computing, the immediate nature of both software and application suggest it may well be worthwhile. In some programming environments where programming application is small, immediate, and short-term, the same logic might apply. But for the main concern of this study, larger and longer lived programme artefacts, the argument does not seem convincing. Indeed, if the relevance of the natural language context is to be accepted, then interest in multilingualism should follow. And psychological study (see Obler and Albert, 1978) indicates as advantages of multilingualism not only the obvious facility in other languages, but increased facility in various language oriented contexts crossing between and outside the particular languages involved.

In his partisan arguing for the essential relativity of all language, Robinson (1975) suggests the model for language quality should be the same as that for art. And that there is only the same ultimate criterion applicable: survival. In fact the situation may well be more extreme, because that criterion for art is not universally accepted. Pragmatic influences take their toll, of course, but even the necessarily transient and fleeting have a place. Poetry need not be literary engineering; need programming - or programming language design - be software engineering? Can we reason to convergence? No. Or not yet. And it looks not soon. Assuming this to be so, the next step is consideration of how to best cope with continuing diversity.

PART TWO

COPING

From INTRODUCTION:

Over all, this exploration of solutions to the problems of programming language diversity is structured with three divisions progressively narrowing in scope.

First is a consideration of the possibilities for reasoning to convergence in programming languages, thereby eliminating the problems of diversity by eliminating the diversity: integration at the roots of diversity. Central here is the question of programming language "quality" and how it might be evaluated. These questions are extraordinarily deep, and conclusions can really only underline the difficulty and importance of the problem. In these discussions scope is generally restricted to matters directly involving computing. As matters seem to lead away to other experience and more general enquiry, this is somewhat arbitrary and perhaps rather unsatisfying. Within the practical context, however, simply establishing the direction of difficulties is sufficient.

Second is a survey of established computing practices and techniques that in some way approach solution to the diversity problems. This a reminder that much that is impossible to totally resolve can still managed with tolerably. The approaches examined are grouped into those seeking "compromise": reducing different languages to one, and those seeking "cooperation": ways of using different languages together. Often the diversity problem has been only a minor or unstated concern of formal developments and studies, so some conclusions can be only informally drawn. Many strategies still have definite advantages in certain circumstances, however, and several deserve to continue being applied. But most strategies also have significant drawbacks that do keep their usefulness limited. Even so, some cooperative practices appear on examination to suggest further development of a more general strategy may be worthwhile.

Third is the proposal for a new general strategy for addressing and tackling the diversity problem. The philosophical problems realised in the first part of the study and the practical directions suggested by the second section do not quite seem in collision. Indeed it seems possible to practically evade problems of diversity as much as any particular programming language design permits, while avoiding any necessary implication of impossible universality. So such success must necessarily be limited, but some increased success is possible. The approach involves minimally collaborative design in programming languages, and necessary support from environments exterior to them. Though grounded in applicable practice, these proposals do not follow through in requiring specific action - there remains much choice. These are design principles for languages and support systems, and could be evolved toward. However, several programming languages are explored, and opportunities for useful access described or modifications suggested. To practically examine the question of the exterior support, two operating systems are also discussed, and the requirements of necessary design established. The importance of new software both to assist and take advantage of this "programming between programming languages" is illustrated with some new programming tools.

So more exploration and experience are indicated, and detailed study for particular programming languages and operating systems design would then be of most concern, as well as more attention to particular questions of performance and efficiency. Such particular work is outside the current scope, as indeed are several other directions suggested for future investigation. Explorations in representation diversity in other fields may deserve investigation next, and the balance of subject matter could be important as well as enticing. The contributions of this study are the mapping out of the vast area of this important subject, the review of relevant past practice, the development of a general strategy for progress, and the directions for its practical pursuit.

CHAPTER III
COPING BY COMPROMISING
REDUCING DIFFERENT LANGUAGES TO ONE

Many strategies that have assisted in coping with programming language diversity still encompass the possibility of some resulting new language or "language-like" system as a compromise between existing languages. In this view, resolution of the diversity problem lies in reducing, albeit only by compromise, diversity itself - yielding a melting pot of linguistic style and capability.

Attacking problems of diversity or indeed multiplicity through compromise requires some optimism. In theory, the grounds for even comparison, as for convergence, are alarmingly unsure. And in practice, feelings preserving or leading to diversity are often clearly strong. Designing a language or language system that makes compromise, and makes it compelling, is not be easy.

Unity is often the cry when diversity threatens, yet the mechanisms of fair play are not clearly understood. Unity and diversity both have advantages and disadvantages, while the latter leads away from convergence, the former might lead nowhere at all.

SECTION I. STANDARDISATION
DIRECTLY ESTABLISHING UNITY

Though not a way to eliminate the diversity problem entirely, programming language standardisation is clearly a way to reduce the diversity possible. The compromise to a standard can dramatically smooth diversity problems within the community of a particular language or language family. And in so doing, that compromise also eases significantly diversity problems between programming languages in general. It is perhaps worth a word about the importance of standards in the wider technical world: pivotal. It is a central element of any mass production both in input component and output product; it was a key element in the industrial revolution; it's an essential element of most systems of interconnecting parts - language being one.

The term "standard" distinguishes one of a programming language family for favour over the others in specification, reference, and use - perhaps even favour to the exclusion of others. The term "standardisation" encompasses a range of activities in establishing this distinction. As might be expected of terms involving distinction and consequent prestige, usage in fact varies: the above definition is an attempt to embrace the common variance. It seems useful to regard even accidental prescription and convergence as standardisation if effective; useless to regard even international certification as standardisation if little heed is actually paid. De facto standards may be strong; international law may be ignored. As Ross (1976) concludes in his practical advice on working towards standards: "Standards exist only to serve the needs of a non-standard world. Standards cannot create a standard world."

A standard may develop in several different ways. Several of the differences can probably be illustrated using two of the earliest languages to have widely recognised standards: Fortran and Algol. In the

case of Fortran, the early standard was simply the Fortran as implemented by IBM for the IBM 704 and successor computers. And the reason was simple, Fortran was created at IBM, IBM was the dominant computing supplier, and IBM ensured wide publicity, distribution, and support. Backus (1978b) gives a review with hindsight of the early history. The sequence of events ensured a language investment that thirty years later still influences hardware and software design. From the view of standardisation, the important points seem to be that it was a narrow model because it had a single corporate source, and that it had impact because of the strong support available to users.

The establishment of standard Algol followed a quite different path. While much consideration of the language was conducted publicly, the design itself was done by a committee representing different interests, and eventually it seems some decision was left with the "editor" of the eventual report, Naur. Naur (1978) and Perlis (1978) relate the history of the development. To the extent that the language of the final report (Naur, 1963) did become an acknowledged standard, it is probably because of two reasons. Firstly the difficulties of diversity were understood and so the need for agreement even if in compromise, and secondly there was importance attached to clear and precise description of the language, hence the adoption of the notation (now called BNF- Backus Naur Form, and still very influential) proposed by Backus. Much of the experience in language standards and language standardisation may be seen as foreshadowed by this early history.

It might appear that the Fortran and Algol stories exemplify opposite poles. But there is common ground because both language standards were attractive to follow because it was reasonably easy so to do, they were accessible. In the case of Fortran, the implementing software was readily available and for computers that were readily available. And when a Fortran language standard was recognised by the American authority (USA SI, 1966), the issue was irrelevant, as most Fortran users were not affected; the Fortran language as implemented was different (Larmouth, 1973, Ryder, 1974, Sabin, 1976). In the case of Algol, the language definition was readily available in a form applicable to computers in general. Perlis: "The Zurich language (and Algol 60 as well) was an invitation to implement a language processor." And in Europe, anyway, an invitation that was accepted. Why Algol was not more widely implemented, especially in America, is not totally understood; there seen to be many pragmatic influences. It was accepted everywhere as the medium for expression of algorithms for discussion and research. The role of standards in this area is clearly important in avoiding the misunderstandings possible with confusing diversity.

There are obviously many reasons to prefer one language to another, ranging from the difficulties of language quality itself to the pragmatics of commercial, political, and social pressure. About such reasons in general it is obviously difficult to reason to compromise in any deterministic way. The problems remain interesting and important, of course, and it seems reasonable to learn about these processes, deterministic or not, especially where experience is available. Historical discussion about programming language development standardisation is available for particular programming languages from a number of sources particular to the language, and more collective histories in Sammet's (1969) great survey, and more recently in the various presentations at the ACM Sigplan History Of Programming Languages Conference Proceedings (Wexelblat (Ed.), 1978), and the subsequent book (1981). In looking at the more general difficulties on progress and compromise in development and standardisation, Bemer's (1969) review of the

"Politico-Social History Of Algol" is a rare exposure of the interplay of the various criteria for design: linguistic precedence, mathematics, psychology, and other contexts.

In as much as sheer accessibility is a factor, however, some progress does seem significant. Obviously even this might have more general overtones, of course, but there is a more tractable side. There are two areas that might be distinguished: work towards portability of language implementation, and work towards formal language specification. As discussed in arguing for looking beyond the portability problem to the diversity problem, the gap between these two approachs seems to be narrowing, if not disappearing in overlap.

Given the diversity in the software and hardware levels upon which programming languages might be implemented, portability of implementation is clearly an important step to general accessibility. So where other pragmatic factors influence acceptance of a language, it should be remembered the technical difficulty blocking actual use of the language can be reduced very greatly. Certainly software for assisting with language development can ease the implementation difficulty significantly, especially where maximal efficiency is not essential. Such software is reasonably common (see Aho, 1980 for a survey), and well used at least in some environments (for the Unix environment, for example, see Johnson and Lesk, 1978, or Johnson, 1980). But at the next level of implementation, efficient code generation, the task is more difficult. There are a variety of methods for such portability, and with each one various advantages and disadvantages: Ganapathi, Fischer, and Hennessy (1982) present a good survey. The central problem has been that the most simple and most portable are generally the least adaptable to the peculiarities of particular supporting environments and so often are inefficient. There are suggestions and explorations for heuristics in adapting simpler strategies to a variety of circumstances (see the overview of the "Production Quality Compiler Compiler" project by Leverett et al., 1980, and Ganapathi and Fischer, 1982, 1985, for examples).

The wide distribution and success of Snobol, for instance, Griswold (1978) does see pragmatically to have been achieved because of the portable implementation. The success of Pascal, too, in becoming so widely used, Wirth (1985) attributes to the early design of a simple recursive descent compiler generating intermediate code. Even when a differing Pascal was recognised by the International Standards Organisation (ISO 1980, or Addyman, 1981) the existence of implementations of the earlier language (Jensen and Wirth, 1976) and none such from the ISO would have contributed to reluctance to follow.

The development of formal techniques for programming language definition and specification has stemmed originally, with Algol, from desire to specify without undesired ambiguity and misunderstanding too easy with use of natural language. However, the potential has expanded, as McGettrick shows in his historically oriented survey of formal language definition (1980). In fact, it should always have been evident since the development of Lisp, a language capable of easily defining itself - at least in a simple version (see McCarthy's summary, 1978b). But later, with the Vienna Definition Language (Wegner, 1972) used for PL/I (ANSI, 1976, and Marcotty and Sayward, 1977) and the use of van Wijngaarden's two-level grammars (van Wijngaarden, 1965) for Algol 68 (van Wijngaarden et al. 1976, and McGettrick, 1978) it became more clear that formal definition of larger languages was also possible. At this point the identity of definition and

implementation also became clear, enabling total definition, both syntax and semantics, as illustrated with the small language by Cleaveland and Uzgalis (1977).

There seem to be two approaches to the practical use of this principle, one the consideration of automatic implementation from more mathematically oriented definitions (see Cattel, 1980), the other simply defining languages in terms of already defined and known languages, as advocated by Pagan, 1974-1980b. It is not really clear what the formal distinction between these approaches is. Pagan has recently (1984) proposed a synthesis, suggesting design of languages along conventional so "understandable" lines, but expressly for language definition. No major programming language has become established with an operational definition, though the advantages appear enticing. The formal definition of the intermediate language, EM (Tanenbaum et al., 1983), by a Pascal programme may be indicative. There are problems yet, of course. There seems to be a difficulty in satisfying three different needs: formal and mathematically tractable definition, implementation suitable definition, and of course human understandable definition. And the last is still a great difficulty. The notation employed in describing Algol 68 seems to many people intimidating and a liability in itself, despite the power and precision of the two-level grammar notation. Algol 68 never became as popular as initially expected, and an important reason might well be the concentration on formal definition and no immediate provision for people (or computers either, there being no implementation mapping). Hoare (1981) seems pessimistic about both the formal definition of Algol 68 and about the natural language of the Ada definition, and argues that such huge size is a severe liability in either case.

Standard establishment is certainly a way to reduce diversity and thus the diversity problem. A standard is, in fact, the antithesis of diversity, within it's own context. It is a difficult subject to reason about because of the many intractable but pragmatically important factors involved, but the growing experience and knowledge in more automatic implementation make standardisation seem potentially more possible. And while the wider question of programming language quality might remain unresolved, it is often possible within the community of one language family to make the compromise required.

However, there are yet problems in mastering even the tractable, as efficiency is still difficult in portable implementation. More generally, the attractions that lead to diversity persist: standardised languages continue to evolve in spite of programme investment. And still, there is the diversity between programming language families: gaps of such width there must be doubt that compromise can bridge them. As long as new and differing notations are attractive, standardisation may seem an impediment. It doesn't really matter whether they are in fact more effective, the problem is not that clear. In the wider technical world standards are pivotal, but where the problems of diversity can be afforded they are often paid. It might in many ways be more effective and efficient to design, produce and maintain only one programming language - or indeed one, say, model of motor car. But as well as many pragmatic imperatives involved, there are also still diverse views of quality, and a continuing desire to explore design possibilities without self-imposed restriction.

SECTION II. MINIMISATION
REDUCING LANGUAGE DESIGN TO A COMMON BASE

In seeking compromise, one general approach often seen attractive involves the concept of reduction, even to some perceived minimality. A minimal programming language need not be an immediate regression to, for instance, an unadorned Von Neumann computing model expressed in binary - not itself a guarantee of compromise! Weinberg (1971), for example, argues the superiority of the PL/I array subscript (an expression) over the Fortran situation (c*v+k) saying that non-existent rules are easier to remember, and similar views are widely stated on a variety of subjects. There are many virtues of minimality, and in seeking compromise amid diversity its virtue is that if no particular view is favoured, all views will be on the same footing. It might simply be by the establishment of a set of common components none of which anyone wishes to do without. In general this proves quite difficult, a similar task to that discussed earlier with regard to the establishment of axioms of programming language design. In practice, however, it has proven useful in agreeing on parts, and in some cases substantial parts, of some language design. Several examples of this approach in practice can be found in programming languages proven quite successful. There are different ways in which minimality can be regarded, and it can be difficult to decide just what minimality means. In leading toward compromise in language design, it usually means avoiding a design aspect where otherwise diversity might result.

Minimality might first arise in the process of design. Consider for instance Algol 60, specifically the absence of explicit provision for input or output. Several reasons might be seen for this absence: the orientation towards descriptions of algorithms for people, rather than implementation on machines; the understanding of lack of understanding about the subject and an attendant inability to conceive of truly appropriate methods; or the simple inability of the Algol committee members to agree on any one method. All appear to have been involved to some extent, according to the histories, (Naur, Perlis, 1978), and the decision taken was then regarded as wise. The resulting non-definition, then, might be seen as compromise by rough minimality: implementations of Algol input/output may differ, but all will still be Algol and thus able to be used for communication - and even in execution with non-linguistic adaptation. Naur: "In the Zurich report it is stated that operations of input and output will be taken care of by means of procedure declarations that are partly expressed in a different language than Algol itself." However, it is also considered that the non-definition of input and output has also been a reason for some lack of success. But it is difficult to certain, and the commercial, political, and social factors involved might indicate otherwise. Algol has in fact been most popular in a context of natural language and notation diversity, Europe; and conspicuously less popular in a context dominated by one language, one notation, and it may perhaps be said, one computing supplier, America.

Minimality might also be an alternative after design. In the case of Fortran, for example, despite the USA SI (1966) standard there was much difficulty in coping with the diversity of Fortran implementations. In addressing the problem of programme portability between diverse Fortran implementations, the approach of Ryder (1974), was of minimality. By surveying a wide variety of Fortran implementations, an intersection set could be determined, and forms the basis of Ryder's PFort Verifier. The verifier is not a compiler at all,

but simply investigates programmes for accordance with the intersection language, documenting differences. With this tool (itself written in Fortran within the intersection), programmes can be kept within the Fortran language widely portable. So by minimising differences between the Fortran dialects, PFort Fortran is a compromise approach to that particular diversity.

Another use of minimality in language design compromise is the limiting of variety at a low level of language structure, that limitation then pervading much programming in the language. A good example of this technique concerns provision for flexible composite data structuring. At the programming level there may be lists, strings, trees, pointers, addresses, and more; at the programming language level there may only be lists (Lisp), or strings (Snobol), or vectors (BCPL), or arrays (Fortran and APL). Of course we know that such a structure is always enough for sufficient computability in theory, and where language design is sufficiently skilful there is often enough in practice. Similarly, it may be held that all these structures are in some sense the same at some "deeper" level. However, it's not clear what this means: we know they can model one-another, but we have no universal frame, and they do have differences seen significant enough to give rise to the design, diversity notwithstanding. This is not new or unique to computing: in mathematics many different structures might be modelled on one, as a graph may be modelled on an adjacency matrix. In cases where the entire language has been planned about the minimality, the flexibility with which it may be employed is very great indeed, and practitioners often claim no need for other structuring primitives. It's not clear at what point a programmer is a virtuoso; it's not clear at what point a programme is a toccata.

A refinement of lower level minimality is the concept of orthogonality in language design. The technique involves the separation of programming language components in various ways so to achieve minimality within each partition, yet multiple functionality through practical use of components from different partitions.

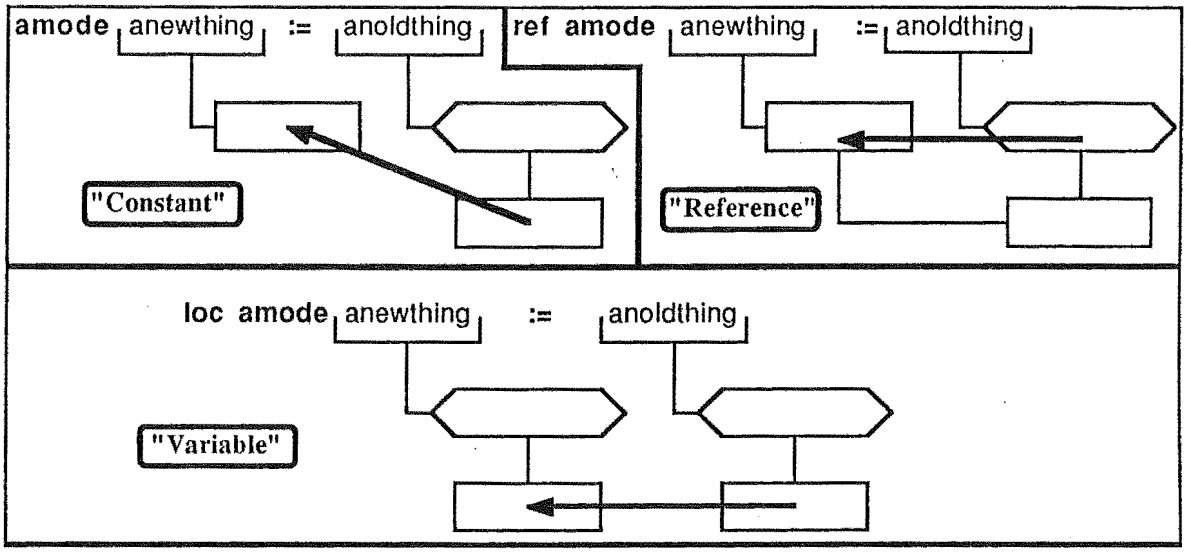

Figure 3.2.1: Orthogonatity in Algol 68 data - Contstants, Refences, Local variables - declaration, creation, and initialisation within the same structure. (Hexagons represent "reference", rectangles represent "value".) [Adapted from Lindsay and van der Meulen, 1977]

This strategy has been described, and the term coined, in the presentation of Algol 68, and that language provides a good example. There is orthogonality throughout the language, but particularly impressive is the clarification involving objects, modes, generators, names, and reference in general. The design encompasses static and dynamic allocation, constants, variables, and structure, then allows integration of such data with the clause and unit basis of control. [The essay proposing a relational basis for data structuring (Biddle 1983a) that motivated this general study was itself inspired by this Algol 68 approach to data.]

It becomes evident very early on in learning Algol 68, though, that the orthogonality has a perhaps surprising drawback. As Lindsay and van der Meulen apologise in their "Informal Introduction To Algol 68": Since Algol 68 is a highly recursively structured language, it is quite impossible to describe it until it has been described" - so explaining why the title of their first chapter is "Very Informal Introduction To Algol 68"! (And "Informal" is used here in a very formal sense!) With the Algol 68 design as motivation, Denvir (1979) attempts to refine the concept as a design principle. While that exploration is interesting in itself, there is not really any design resolution, and the von Neumann machine, for instance, might be seen as quite orthogonal. Where the capabilities desired generated are well understood, and where a smaller number of capabilities that generate them are understood, this is a good vehicle for compromise. But it is not always easy to achieve. Ghezzi and Jazayeri (1982) list criticisms of Pascal's lack of orthogonality in comparison to Algol 68, mostly concerning distinctions made about different data objects in various situations, then quote Wirth (1975) on Pascal: "If we try to achieve simplicity through generality of language we may end up with programs that through their very conciseness and lack of redundancy elude our limited intellectual grasp.... The key, then, lies not so much in minimising the number of basic features of a language, but rather in keeping the included facilities easy to understand in all their consequences of usage and free from unexpected interactions when they are combined." So Ghezzi and Jazayeri decide that Pascal is not orthogonal, but is simple.

Though "simple" is not necessarily an understood quality. In answer to the criticism of Hoare that Ada is too complex (Hoare, 1980), for instance, Ichbiah replies (1984): "When you judge something, the complexity is not in the details, but in whether or not it is easy to infer details from major structural lines. From this point of view, I consider Ada to be very simple."

Restricting language level variance might also be at the root of much success with "typeless" languages such as BCPL (compared to that of its typed parent, CPL). While the approach clearly has attractions of implementation and freedom in machine dependent contexts, the absence of typing eliminates an entire level of diversity concern.

In minimising a programming language, there is a great range of austerity possible. It is minimality as compromise in eliminating a single superfluous isolated part of a language - the "repeat" of Pascal is one often suggested (Sale, 1981). At the other extreme it is minimality as compromise to orient an entire programming language about a single concept. In this latter category, a clear example is Lisp. Lisp is fundamentally oriented about the structure of "list", as defined by few simple primitive operations. This orientation transcends the idea of minimality in "object" as Lisp programmes are themselves are seen as lists

and can be so manipulated and even implemented: it transcends the concept of "data". The minimality here was achieved by design, though explicit compromise perhaps played a lesser role than accident, as McCarthy (1978b) relates that it was for some time intended to work above "S-notation", that now typifies Lisp, with an "M-notation" that was "intended to resemble Fortran as much as possible".

However, McCarthy points out that the minimality has had great benefits: implementation, verification, and education. In total, Lisp has been quite successful and the minimality aspect may well be a fundamental reason why. Many applications commonly made upon Lisp do not seem to require the Lisp "list" paradigm, but do yet build upon it as a simple, and portable, foundation. In this case, much difficult programming, little "syntactic sugar" is required, but a raw though usable computational model is suitable. With little fuss, this is what Lisp provided.

As shown, minimal language or minimal language componentry can often be seen as a route to agreement, to popularity and success - and minimality might be employed in an one area of particular difference or in some sweeping application of varying severity. It is an important approach, and by avoiding possible diversity can be successful without extra production burden: silent films need not be dubbed. So in theory, minimality is attractive. And in practice, it is useful, but also has severe practical drawbacks. Some advantages of minimality, real though they may be, can be subtle - perhaps unhelpfully subtle. Unfortunately, minimal language is often seen as scope for improvement through addition. So often the possibility of compromise through minimality might be lost by later extension. And so Lisp, for one example, suffers greatly from differing variations, in practice, if not in theory, greatly reducing the fruit of compromise. In some cases this might seen worthwhile evolution, in others perhaps needless ornamentation. Hoare: "Algol 60 was indeed a great achievement, in that it was a significant advance over most of its successors" (Gries, 1981).

In either case extension is a movement away from the minimal. In general, minimality is a way to compromise, but the decision procedure is difficult. What to keep and what to throw away? Even when minds are made up, they are often later changed. And even redundancy might have some merit.

SECTION III. MULTIPLICATION
EXPANDING LANGUAGE DESIGN TO ENCOMPASS DIVERSITY

Though in essence an opposite approach to the minimality that can lead to compromise, design multiplicity can be seen as an approach to compromise. In language design, it might be argued, it is the presence of the attractive, rather than the absence of the repulsive, that is of paramount concern. In theory, multiplicity has a similar problem to that facing minimality: it's difficult to know where to stop. In practice, however, this approach too has proven somewhat successful in finding compromise between differing ideas in programming language. In some cases, two or more previously established sets of ideas may be involved: these might be called "combined" programming languages. In other cases, many previously articulated ideas may be involved: these are typically called "super" programming languages. The latter name is misleading, however, as the resulting language is not necessarily "superior" to any component language in fact, and are not really "above" them either. "Multiple" should cover all these approaches, and complement the earlier discussed "minimal".

Combinations of exactly two previous languages are not very common, though it has recently become especially popular to mate languages with the logic capabilities of Prolog (Clocksin and Mellish, 1981). One such is Logicon (Lapalme, Chapleau, 1986), an integration of Prolog and Icon (Griswold and Griswold, 1983). The language is really Icon, in fact, but with Prolog implemented within it and accessible through function calls. The two languages are integrated internally, sharing the same data structures, but syntactically Icon of course does dominate. The language appears to well address a useful application area, coupling the advantages of the two partner languages: logical recursive searching within flexible character manipulation on input and output. Icon provides the conventional frame, and Prolog is called on whenever necessary for it's more specialised services.

Some languages have borrowed features from others, of course (borrowing Fortran input/output was once common), and sometimes new language constructs are proposed combining those of separate languages. Sethi (1981), for example, suggests a type expression and declaration structure specifically combining that of C and Pascal. In local settings the demand and popularity of such languages might well be high, but these are clearly outweighed by the limitations of such local tailoring, and wide popularity seems unlikely.

More common are attempts to encompass several languages into one. In some cases, this is simply to satisfy a local need. Takeuchi, Okuno, and Ohsato (1983), for instance, report that as their reason for developing "Tao: A Harmonic Mean Of Lisp, Prolog, And Smalltalk" as including the provision of particular aspects of these languages without having acquire three different sets of language software.. More well known are the attempts to encompass more general purpose language paths into one. The first such languages to be so designed were probably CPL and Pl/I, and more recently, Ada might be seen as the same tradition. All introduce new ideas as well as covering old.

The history of CPL (Barron et al., 1963) is particularly interesting, not only because it was one of the earliest multiple languages, but also because of its later evolution. The designers sought to combine

Fortran, Algol, and several less widely known dialects of "algebraic" language in use in Britain at the time. It was implemented, well used, and so was moderately successful - Sammet (1969) credits it with influence on the more commercially targeted PL/I. For many applications, however, it seems to have been thought the CPL was too complex, and a smaller descendent language, BCPL, followed. The feeling was that CPL was too much an assortment of different parts and that a single small design was superior (Richards, 1969, 1971). So in this case, some people clearly preferred minimality to multiplicity. CPL is now extremely rarely used; BCPL and its even smaller offspring B (Johnson and Kernighan, 1973) are probably used more. Much more widely known now is their offspring C, a larger language and again partially so by inclusion of some ideas introduced in Algol 68. What has been learned? Perhaps that minimal languages become better established - perhaps simply because of ease in implementation -but then naturally evolve into larger languages.

PL/I began as a projected successor to Fortran, but quickly became an attempt to cover - and expand upon - several preceding languages: Fortran, Algol, and Cobol included (see the Radin's history of PL/I, 1978).

```
TST:    PROC OPTIONS(MAIN);
        DCL 1  D,
                2       HEAD CHAR(60) INIT
                (' CATEGORY   FICTION   TECHNICAL  PERIODICAL   GENERAL'),
                2       A(0:9),
                        3       C FIXED(1),
                        3       B LIKE SALES;
        DCL 1           SALES;
                2       (FICTION, TECHNICAL, PERIODICAL, GENERAL) FIXED BIN;

        GET FILE(SYSIN) LIST(B);
        SALES = 0;
        DO J = 0 TO 9;
                C(J) = J;
                SALES = SALES + B(J);
        END;
        DO J = 0 TO 9;
                B(J) = 100.5 * B(J)/SALES;
        END;
        PUT FILE(SYSPRINT) EDIT(D)
                        (SKIP, X(5), A, 10(SKIP, 5F(12)));
END TST;
```

Figure 3.3.1: Programme illustrating PL/I's heritage, in this case showing influence from Cobol (data structuring with level numbers in declaration), Algol (control structuring with free statments in blocks), and Fortran (input/output lists with formats). [Programme adapted from Weinberg, 1970]

In this, traces of the ancestry can easily be seen in language use, and commercial success was achieved, though this was somewhat preordained by its sponsors. Unlike CPL, PL/I was designed and even more developed by commercial interests. And IBM had both the motivation and ability to support the language strongly: they had themselves developed Fortran but seen it become common currency, been a prime supporter of Cobol designed to span suppliers, and were the largest manufacturer and supplier of computers and computer services in the world in general and America, the largest market and IBM's home, particular. This would make a strong background for any language. Dijkstra compares programming in PL/I to "flying a plane with 7,000 buttons, switches, and handles to manipulate". But then also says that because

of the IBM support that "PL/I was doomed to succeed" (1972). PL/I had much investment, and did indeed succeed. However, few outside the sphere of IBM were swayed: one line of software passed through General Electric and Xerox, then Honeywell, and now Prime. The non-universality of PL/I need not bother IBM; moreover, the commercial implication of diversity should be remembered. A barrier to communications and conversions is a conservative force. Whether this seems commercially bad or commercially good depends largely on point of view.

IBM's PL/I advertisements of 1968 asked: "Can a young girl with no previous programming experience find happiness handling both commercial and scientific applications without resorting to an assembler language?" The multiplicity of PL/I is successful in that it is possible to write commercial, scientific, and "systems" programmes. However, the multiplicity is also at the root of common complaints about PL/I. A programmer often finds there is too much to learn; that it is unclear which of several similar devices is most suitable for any particular application; that some language components don't fit together well. An implementor finds the language a very large project, some compatibilities difficult to achieve, and is annoyed that many difficult to implement subtleties may never be used. An analyst is aware of all these difficulties, and is hesitant to trust more than a fragment.

In fostering the establishment of a high level language (now Ada) to suit its perceived need for imbedded computing applications, the American Department of Defense hoped to avoid the problems of diversity, "proliferation" so-called. The design process, most significantly, was a selection of competition by teams, rather than a committee effort. However, the earlier specification was still a committee process, though iterative and public, and that, along with the nature of prestige and power about the enterprise were not conducive to production of a small language. In fact, smaller existing languages were earlier rejected except as a basis upon which to build. And it is the size of Ada, now complete, that is most severely criticised. There is a conviction that the language is simply too large - that too many features are involved for ease of learning, ease of use, ease of understanding. And in this particular case, the intended application area of the language suggests great concern is appropriate. The warning of Hoare (1981) was unequivocal: "The original objectives of the language included reliability, readability of programs, formality of language definition, and even simplicity. Gradually these objectives have been sacrificed in favor of power, supposedly achieved by a plethora of features and notational conventions, many of them unnecessary and some of them, like exception handling, even dangerous. We relive the history of the design of the motor car. Gadgets and glitter prevail over fundamental concerns of safety and economy."

In this last case, the size itself, the multiplicity itself, is seen as a problem, not a solution. Like PL/I, though, Ada is a sponsored effort, "doomed to succeed", and it may be noted that the sponsoring organisation is one of few in fact even stronger than IBM. Multiplicity can sometimes be a way to compromise, clearly. However, firstly, not all multiples seem reasonable to make - design principles can clash at several levels. And, secondly, the very process of multiplicity itself seems troublesome. Larger more unwieldy constructs cannot but result, and, should the approach be applied continually, can worsen beyond tolerance. Work in exploring languages specifically bringing together two or more frameworks is continuing. Recent research emphasis has been on explicitly bringing together control orientations as different as those in functional and systems programming - Hallpern (1986) introduces a selection of the recent efforts this area.

This research is continuing, and results indicate that development is certainly worthwhile where the differing aspects of multiple languages are actively going to be used. Feelings are still strong that language frames do indeed matter, and there is also reported the feeling that programming in multiple paradigms can provide the wider vision that Obler and Albert (1978) suggest in the natural language case.

SECTION IV. EXTENSION
ENCOMPASSING DIVERSITY BY EXTENSIBILITY FROM A COMMON BASE

A form of compromise offering still considerable flexibility is programming language extensibility. In this approach, compromise is essential in the establishment of the firm kernel for the language and in a basis for an extension capability within the language itself. Almost all programming languages might be considered extensible with respect to some computing model, but this view is eliminates any distinction. Many programming languages might be considered capable of extension simply through programme components written in the language already existing, but this is clearly a very limited form of extension. In practical use of the terminology, an extensible language is usable with no extension, but requires no very considerable effort to extend, yet offers when extended new capabilities and new structures with greater applicability or some differing view.

Extensibility has proven popular in several language developments over some time. The term is probably not as popular now as it once was, perhaps because of differences about just what it implied. A notable survivor of the idea, especially in influence, is Simula, particularly in its influence on object oriented programming and programming languages. However, the idea was extant before the term and is popular now even when the term is not used as much.

Symbol Processing

Probably the earliest languages most reasonably considered extensible were those originally designed for symbol processing and thus very flexible in dealing with programme text itself. In Lisp, for example, the base language is obviously quite minimal, but as the only control and data structures both involve lists, so programming involving lists can thus quite further develop the language while fitting well in. It's a language ideally suited to discourse about itself. Similarly in Snobol, facilities exist for both sweeping or more integrated change in the language, though perhaps without much elegance. Most generally, the "exec" facility allows any string to be "executed" as a Snobol statement, and the string manipulation capabilities of Snobol are thus easily capable of changing to a new syntax implemented by the old. More specifically, the "op syn" feature allows the interpretation of any operator ("+", "-", etc.) to be performed as required when the operator is actually invoked. In this way a programme may stay with the syntax of the language and yet turn it entirely to different semantic application - from integer to matrix arithmetic, for example. Such extensibility is quite easily achieved in languages, like Lisp and Snobol, that necessarily involve at least some implementation by interpretation, and so where syntactic and semantic structure are available for manipulation at execution. There is great flexibility and subtlety in such extensibility, though perhaps too great a power in changing the language - too general a tool.

Language Preprocessors

A simple alternative approach to extension is the coupling of a base language with a flexible yet separate and easily implemented preprocessor. In this way, language extensibility is available to programmers, needs little affect base language design, and can always be done without if needs be. "The idea of macroinstructions is not new" is the opening sentence of McIlroy's 1960 proposal to augment programming languages with macroprocessors. And of course the use of macroprocessors with and as part of assembly language is venerable. But it was some time later before the coupling became tighter, as suggested by McIlroy, for higher level languages. One method was to establish for use with any programming language a very general macroprocessor for extension, such as those proposed by Waite (1967) or Brown (1967). The other method was to integrate a suitably designed macroprocessor with a particular programming language. The first widely used language to use the approach was PL/I; today the approach is widely known in the C language and associated C preprocessor. It is common in both languages for the preprocessor component to be used for tasks other that language extension - notably software reconfiguration and parameterisation - but language extension is certainly practical. In PL/I, the preprocessor component is actually a defined part of the language and is implemented along with the language; in C the preprocessor is technically simply another piece of software often used before the C language software. Use of the PL/I preprocessor is probably not extensive and certainly in some environments is quite rare. The first commonly available language software (IBM/360 "level F", 1972a) was quite commonly thought very inefficient (see Wortman, Khaiat, and Laskar, 1976), and especially so with regard to the pre-processor, so initial usage was small. While there have been advocates of extensively using the system to tailor PL/I to specific applications (see Schwartz, 1972, for example), and while more modern software is quite efficient, it seems some paranoia may linger.

In C use of the preprocessor is quite widespread; it is probable that more programmes make use of it than not. However, the primary use is to name constants or manifests, and the secondary use is for simple macro functions: only infrequently is it actually used to effect what might be called language extension. Notable examples of this, though, do abound where some other language is a favoured context. For example, some Unix software was written by by someone who felt sufficiently strongly opposed to some C syntax to "change" it. The programmes are pre-processed with a set of macros that create a look similar to Algol 68 - the programmes appear at first not to be C at all.

```
#define   REG       register
#define   BOOL      int
#define   IF        if(
#define   THEN      ){
#define   ELSE      } else {
#define   ELIF      } else if (
#define   FI        ;}
#define   BEGIN     {
#define   END       }
#define   WHILE     while(
#define   DO        ){
#define   OD        ;}
#define   ANDF      &&
BEGIN
        REG BOOL slash; slash=0;
        WHILE !fngchar(*cs)
        DO      IF *cs++==0
                THEN IF rflg ANDF slash THEN break; ELSE return(0) FI
                ELIF *cs=='/'
                THEN  slash++;
                FI
        OD
END
```

Figure 3.4.1: C preprocessor used to create Algol 68 - like disguise for C. [From the Unix "Shell" - Bourne (1978)]

Common approaches have usually involved either a "universal" macroprocessor, or one especially for and integrated with a particular language. More recently Brown (1980) has suggested a union of these two forms, a single macroprocessor, Supermac, but below surface level, with surface level facilities specific to and syntactically integrated with a particular language. Brown and Ogden (1983) show a Pascal model. The idea might be appealing to language implementors, who could add the facility without being concerned with the details and could integrate the syntactic level, and to programmers, who could find a familiar facility across several languages, but fitting in well in each. Brown reports that the idea has not been yet widely taken up, perhaps because it would be most usefully applied with a degree of coordination simply rare.

Another interesting recent effort is that coordinated by the British Computer Society to develop and establish a standard macroprocessor to facilitate extensibility of Cobol (Triance and Layzell, 1985a). It is argued, and supported by a survey, that though many Cobol users do use a standard version of the language for reasons of compatibility, many are not satisfied with the language, and different users have different complaints. The idea of a "standard" extending macroprocessor, analogous to a "standard" language, seems to be new, and perhaps the politics of "standards" are such that it has a better hope of success than more isolated efforts. Layzell (1985) reviews related past practice, and Triance and Layzell (1985b) elucidate some design principles drawn from that past and the very recent experience. Many of the principles concern the Cobol case, but the foremost conclusion is that macroprocessors for extensibility and hence compatibility should be designed with a particular language and context in mind. This, they see, is the prime assurance that the facility will be widely used, and the benefits of lower level source compatibility realised.

```
#(ds, Hanoi, (#gr, N, 1,
(#(cl, Hanoi, this, #(cl, the other, this, that), ##(su, N, 1))
#(ps, from this to that)
#(cl, Hanoi, #(cl, the other this, that), that, #(su, N, 1))),
(#(ps, from this to that)))))
#(ss, Hanoi, this, that, N)
#(cl, Hanoi, 1, 2, 3)
```

Figure 3.4.2: Trac subroutine and call for "Towers of Hanoi" programme.
[From Cole, 1981]

The limiting case of this extension by macroprocessing would be the programming language that is essentially entirely macroprocessor: alternatively a macroprocessor sufficiently powerful to be called a programming language. Such is Trac (Mooers, 1966), for example, and the simplicity and generality are impressive. However, the surface form of the language is rather daunting. As Cole says in his text on macroprocessors (1981): "The Trac language is not immediately readable to the average layman (this is probably the greatest understatement in this book)." The problem seems possibly and simply that such total dependence on only one structure becomes too confusing, the same problem that Lisp has with lists and parentheses, and packages such as Gasp (Pritsker, 1974) built on Fortran have with subroutine calls. There are very few semiotic clues to the structure. However, there is still great attraction in the approach, and continuing development. A more recent language, Forth (see James, 1980), has become quite successful for applications on very small computers by its strict need for only the most tiny implementation (see Harris, 1980). Forth is not macroprocessor based like Trac, but uses a threading approach to achieve a similar degree of extensibility upon a tiny base. While many programmers criticise languages with such small bases as unavoidably cryptic in applications of large size, the languages often do have devotees convinced they are the ultimate answer.

Integrated Extensibility

The potential for confusion, however, affects more than the application programmer. In implementation of such generally used syntax, the difficulty of identifying the user's intention, never trivial, is compounded. Formally this might not seem to matter, but it is of practical importance in both possible outcomes of programming language use: in error diagnosis clarity, and in translation efficiency. Both can be adapted even to identify particular canonical uses of syntactic extension of course, but so is the utility of such extension skewed. At a deeper level, the idea of extensibility is most important in the general view being taken of data and type. In Simula, for example, an aspect of the class concept enables operations associated with particular classes, and definitions of that operation within the context of that class. This enables some separation of concerns without which extensions of any scale become bewildering - a class is a context of concern (see Palme, 1974, for example, or Marlin, 1976). Birtwistle's Demos (1979) is a good larger example of Simula extensibility, offering a context for discrete event modelling within Simula with the feel of a more specific but harmonious language.

The degree to which the language would or could police such extensions, however, was still limited. In Pascal, first of popular languages to focus on the "data" concept, user defined types became usable in many ways hitherto exclusive to types "built in". Some languages, Modula-2 for example, have developed beyond this in allowing "encapsulation" involving data and procedures together to provide an access to such data only through such procedures so to insist on maintenance of new levels of data abstraction. Some further allow redefinition of infix operators, as did Snobol, to further abstract any difference between types - Ada provides such a facility.

The commitment to extensibility is even deeper in Smalltalk. In Smalltalk, the Simula class concept is developed into a total (though hierarchical) object orientation. In this context not only data but control are encompassed by the object and message discipline, and so both data and control can be flexibly extended. In Pascal or more generally Ada with its "generic" operator capability, new data types can be defined upon existing types, ultimately on some primitive types, and the language, and thus the programmer, can deal with all in (nearly) the same way, syntactically and semantically. Some earlier languages attempted to apply similar principles directly to control flow (see Prenner, 1973). But in Smalltalk, because of the totality of object orientation, new structures of data or control can be defined upon existing structures, ultimately on some primitive structures, with the same generality and flexibility.

```
Built-in control conditional structures:
someCondition ifTrue:       [somethingToDo]
someCondition ifFalse:      [somethingToDo]
someCondition ifTrue:       [somethingToDo] ifFalse: [somethingElseToDo]
Defered "Block":
[somethingToDoLater]
```

```
Method "case" easily added to class "Number":
control
case: alternativeBlocks otherwise: aBlock | |
                            (self >= 1 and: [self <= alternativeBlocks size])
                                    ifTrue: [↑(alternativeBlocks at: self) value]
                                    ifFalse: [↑aBlock value]
```

```
Advent "case" structure:
someExpression case: (Array
with: [case1]
with: [case2]
with: [case3]
...
) otherwise: [somethingElse]
```

Figure 3.4.3: How an Algol-like numeric "case" statement can be added to Smalltalk-80 - using simple built-in conditional structures, "Blocks", "Arrays", and an additional "Number" method. [Adapted from Deutsch, 1981]

Extensibility seems a very advantageous ability, but there are limits to the success with which it may be employed, and limits to the success it may achieve where employed. Firstly, there is a difficulty of implementation. To offer the same linguistic level of discourse, a language extended to that level cannot match the efficiency of a language offering that level directly. And the more extensibility offered, the more flexible an implementation strategy is required. In practice the flexibility might mean, for example, the difference between interpretation and compilation. Secondly, there is a difficulty of application. Where a language itself provides little, the programmer must provide more, and that may be an unwanted

responsibility. Although optional frameworks might be offered, the language frame is then less helpful - and a diversity problem between such frameworks arises.

Language extensibility is very useful and important, nevertheless. In language practice, the adaptation to model other familiar facilities can be welcome as a mnemonic and a documentary aid. In language theory, the flexibility is an economical way to explore new facilities and so continue with and contribute to innovation in language and programming design. In a summing up on extensibility, Standish (1975) decided it was clearly worthwhile because of the flex, but restricted because of the difficulties in controlling potential complexity. He describes extensibility as "do-it-yourself" language, with the attractions and repulsions thereto appertaining. As a compromise approach to resolving problems of diversity, it is useful - perhaps especially in prototyping - but no panacea.

SECTION V. ABSTRACTION
ENCOMPASSING DIVERSITY BY ABSTRACT PROGRAMMING

The problems of programming language diversity and indeed the problems of any particular programming language are often sufficiently bothersome and frustrating to suggest abandoning programming languages altogether and looking for something better. This thinking can sometimes be seen to lead to a programming framework separate, beyond, and prior to that of programming language: perhaps "praeterlanguage" might be the appropriate name. Such frameworks have proven very popular and widely advocated in recent years, though the specific benefits are widely contested. Several approaches are common, and while all regarding language as the later and secondary step they themselves differ greatly.

"Pseudo-Language"

Perhaps the best known such framework is that called "pseudo language". MacLennan (1983) credits usage to Wilkes, Wheeler, and Gill's 1951 "The Preparation Of Programs For An Electronic Digital Computer", but the "pseudo-code" discussed there it seems would now be recognised as "programming language". The current usage seems to indicate the more idea of controlled vagueness advocated and exemplified by the approach of Wirth's influential "Program Development By Stepwise Refinement" (1971a), and the subsequent "On the Composition of Well-Structured Programmes" (1974). As for the notational approach itself, something similar has probably been in use as long as computing has been done. A pseudo language is very much what any person makes of it: it is intended to be a flexible and vague notation for describing programming or computing without the requisite exactness of any actual language. A pseudo language is not normally either implemented or even formally defined, though often suggests in whole or in part programming languages that are.

Pseudo language is probably best seen not so much as successful but rather as inevitable. The vagueness and indeterminacy of such an approach is very great, and perhaps for this reason is more able to play the same role with regard to thinking about computing as the vague framework of natural language does to thinking in general. So in addressing the problems of programming language diversity, pseudo language might well be seen as a compromise. In the common practical approach to using pseudo language, all programmes might be written in pseudo language as a primary step, then "translated" to a programming language as a secondary follow up. This is the approach suggested by Wirth, and also in Dijkstra's very influential "Notes on Structured Programming" (1972). Language differences might thus be spanned by translation into specific programming languages as necessary. Clearly there is here some truth, and the task of implementing some programming into new language might be speeded by availability of its pseudo code source rather than a programme in some specific language (particularly a less well known one).

The problem, however, lies in the wide latitude afforded a pseudo language, and the "cultural" assumptions that tend to be made. Firstly, it is often not a straightforward or even deterministic task to translate a programme from pseudo language source to programming language target - even when both are well understood. Secondly, the bounds of pseudo language are wide indeed, and vague to boot, and the

pseudo language used by a quite competent Lisp programmer may in fact be little use to the expert practitioner in Cobol. Pseudo language is useful thinking tool, but less so a communication tool - it is too limited by context. The ultimate limitation is a pseudo language sufficiently formally defined to permit implementation (see Robillard, 1986, for example); the reality of it being another programming language is then clear.

"Design" Languages

A related concept to the pseudo language appears to be the specification language. The similarity seems to be the level of the language, a step above the lower levels of programming languages of concern, but the formality is much greater. The trend apparent recently is for such languages to actually be specific to a particular programming language. Many such "program design languages" (or PDLs) have been designed for Ada in particular (see Hart, 1982, and many other following papers), and a number are sufficiently formal to be implemented - providing a first step to Ada programming. And with such formality, the lack of generality becomes clear. The PDLs are very Ada oriented, so structures that correspond well with those in Ada are well catered for; other structures are largely ignored.

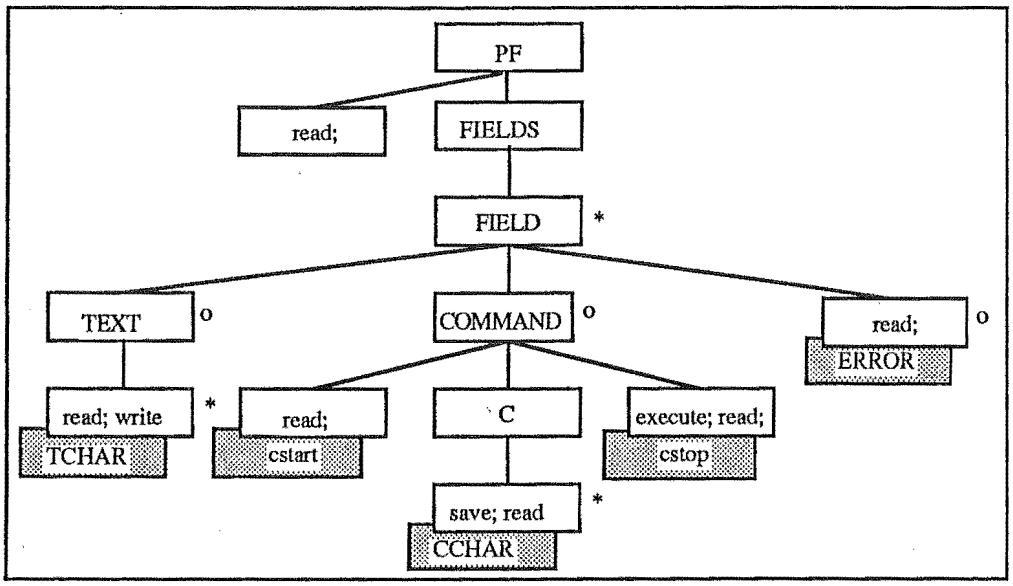

Figure 3.5.1: Jackson Structured Programming diagram, control superimposed on data - (Asterisk represents selection, circle represents iteration). [Programme is VDU controller; adapted from Sanden, 1985]

More specific than pseudo language, and again used in a similar, if more formal way, are some highly structured, often diagram oriented, programming methodologies such as advocated especially by Jackson (1975), though there are other similar methodologies. These are all mostly concerned with commercial programming and eventual Cobol realisation of programmes. However there it is claimed that the approach is general, and should be used prior and in preference to "programming languages": see Jackson, 1980, for instance, for an almost dogmatic methodology. Sanden (1985) discusses systems programming using "Jackson Structured Programming" (JSP), claiming "JSP represents a systematic method for

designing a correct and efficient program from the elements of any language". In such methodologies, diagrams are used to represent data structures involved, and this basis is used to develop control structures for the desired programme - again typically represented diagrammatically. The diagrammatical method is seen as primary and is intended to be translated into a programming language as a secondary step. Jackson argues that the methodology is very helpful in learning to programme and in actual day to day programming - especially as a consistent methodology to be applied by several programmers in common enterprise.

While the subject normally associated with this methodology is commercial programming, it does provide for all the normally powerful computing facilities and is hence all-applicable. And while the methodology clearly sees the approach as an assistance in reducing the problems of diversity, it is not really much of a compromise. The promise is great, but the artefact of the methodology, diagrammatical code, is similar to a structured Von Neumann model, and actual translation into diverse languages might clearly be difficult in practice. Moreover, in an application area where the computing model is substantially different to that of a methodology, the difficulties in specification of programmes would probably prove quite unreasonable.

"Guarded Commands"

There do exist computing frameworks claimed prior to programming languages that are more well defined still. Probably the most well known approach is that of Dijkstra, "Guarded Commands" (1975) and further refined and advocated in Dijkstra's "Discipline Of Programming" (1976) and Gries' "Science Of Programming" (1981). Dijkstra's method evolved in investigation, formal analysis, and development of programmes from principles of mathematics. While mathematics itself has the truly fundamental role in the approach, computing is explicitly applied in the notation known only as "guarded commands". Dijkstra sees "guarded commands" as a minimal computing calculus, knowing that they are sufficient (as are most) and feeling that the little structure is also in fact necessary (and so avoids the tractability problem he associates with unrestricted branching). The notation is sparse, symmetric - even reversible. It is well suited for some programmes: for instance, those able to well use multiple assignment and other forms of parallelism. But not entirely minimal, and yet also not especially well suited for many common programming situations: no "for" loop for example.

Dijkstra advocates the formal development of programmes, and widely demonstrates his methods While formal, the methods are not deterministic or deductive, and so a process results similar to that in the calculus of indefinite integration. In so developing programmes, Dijkstra advocates the use of no particular programming language, indeed of no programming language but the "guarded commands". The notation is nowhere publically detailed in full but the components well known are well defined and it seems there is no reason why the "language" could not be implemented itself. It may be that the notation is seen as not yet complete and so reluctance to cast in stone is understandable. Asked about implementation, Dijkstra responds that this would lead to an unwise temptation to "test" programmes in the notation, instead of relying on their formal development. The canonical approach advocated is indeed to develop programmes formally from mathematics to "guarded commands", and then to proceed to a secondary translation to some programming

language where actual implementation seems desirable. Again, Gries: "one programs into a language, not in it." "Guarded commands", alas, are a programming language - or indistinguishable from one. And, as do all languages, show a bias to an appropriate application domain. As such, the approach is again limited in potential success as a solution via compromise to programming language diversity: the notation addresses some problems, but not all, and is not more demonstrably fundamental than other notations.

Dijkstra's Guarded Commands:

```
i, j, k := 0, 0, 0;
do f[i] < g[j] →  i := i + 1
[] g[j] < h[k] →  j := j + 1
[] h[k] < f[i] →  k := k + 1
od
```

PL/I:

```
i = 0; j = 0; k = 0;
LOOP:
        IF F(I) < G(J) THEN DO; I - I + 1; GOTO LOOP; END;
        IF G(I) < H(K) THEN FO; J = J + 1; GOTO LOOP; END;
        IF H(K) < F(I) THEN DO; K = K + 1; GOTO LOOP; END;
```

Fortran:

```
22      IF (F(I) .GE. G(J)) GOTO 22
        I = I + 1
        GOTO 20
22      IF (G(J) .GE. H(K)) GOTO 24
        J = J + 1
24      IF (H(K) .GE. F(I)) GOTO 26
        K = K + 1
        GOTO 20
26      CONTINUE
```

Figure 3.5.2: Programme fragment written in "Guarded Commands" - and translated into PL/I and Fortran. [From Gries, 1981]

All these methods of "abstract" programming might themselves be paths to resolution of diversity problems, subject to an agreement on context. Diversity, though, is an absence of just that agreement - and these methods offer only compromise on one level: the diversity problem will occur recursively with the "abstract" languages themselves. Even the ubiquitous flowchart imposed language-like constraints when used within strict guidelines often suggested (see Chapin's 1970 tutorial on the Ansi standard, for example).

The practical application of any language "beyond language" will also have problems. The maintenance of programme texts in parallel is often itself a nuisance, and the handling of primary and several secondary forms of programme text - all current - would greatly accelerate the nightmare of administration. Beyond, it is not clear what is beyond language, or what that might mean. All that has just been discussed are yet more languages - useful yes, but languages all the same.

SECTION VI. CONCLUSION

While all these aspects of compromise on a new programming language approach have had practical advantages in limiting or combating the problems of programming language diversity, the success has never been overwhelming. The approaches detailed have other - in most cases more important - goals, and in regard to these goals, success has of course often been more forthcoming.

Beyond the limited success of any of these methods loom further severe practical problems inherent in all the compromise approaches. Even were it possible to compromise on some new language, there is already an enormous number of existing programmes in existing programming languages - and a world of investment of time, energy and experience in the existing languages. Even were it possible to commit to compromise, this backlog is staggering: in collecting, organising, and translating. Even could this be done, could unity be then maintained sufficiently well? And what then, if yet a better framework were presented?

	Advantages	*Disadvantages*
Standardisation:	• Helps avoid unintentional and less determined diversity.	• Difficult to establish wide agreement on detail.
Minimisation:	• Smaller definition can ease portability and so foster wide distribution.	• Not able to cater to many particular application areas. • Eases further diversification.
Multiplication:	• Excellent for applications using many of the integrated components.	• Overly complex for applications not needing many components. • Difficult to implement and maintain.
Extension:	• Good for applications wanting wide use of very specific or uncommon structures. • Good for experimentation and prototyping.	• Potentially limited access to application oriented facilities. • Support environment not tailored to any application paradigm.
Abstraction:	• Good if abstract language well matched with application and concrete target languages. • Can avoid irrelevant implementation concerns .	• Inconvient if poor match between abstract and concrete contexts. • Parallel programme maintenance. • Can defer realisation of pragmatic restrictions of implementation.

Figure 3.6.1: Summary of particular advantages and disadvantages of individual methods of compromise on a single programming language.

All the aspects of compromise on a new system have attraction and show some practical success. Each method offers advantages for particular situations. Even the general problem of software investment can be discounted in the particular situation where the software needs to be largely re-written anyway. But the general success of compromise is too limited, the promise too little, for the commitment ever to be very widely made and widely kept.

CHAPTER IV
COPING BY COOPERATING
USING DIFFERENT LANGUAGES TOGETHER

Some strategies that assist in resolving problems of programming language diversity immediately accept the continuance of differing environments and so seek less compromise, but more cooperation. In this view, resolution of the diversity problem lies not in resolving diversity at all, but in cooperation between contenders - a mosaic of linguistic offerings.

If seeking compromise in establishment of a new language schema seems based on optimism, perhaps seeking mere cooperation between uncompromising languages may seem grounded in grim pessimism. Perhaps agreement just cannot be reached - perhaps language communities would rather steer outward than inward.

Yet, this too presents the optimist with something. Programming languages are very young, and our understanding and experience with them is limited. New programming languages still spring up easily, and new ideas continually demand attention. It may seem foolish to scatter in all directions, but how much worse, how limiting, to compromise too soon - if cooperation is a reasonable alternative.

SECTION I. TRANSLATION
AUTOMATIC CONVERSION OF PROGRAMMES TO OTHER LANGUAGES

Whether attempting to cooperate between language environments or indeed in choosing to compromise on a single language, a process of great importance is translation between languages. In natural language cooperation and compromise this is also an extremely important process, and the reasons are similar. Natural language translation done "manually" is a task requiring much skill in both languages, and knowledge of the subject of discourse. Particularly where a wide cultural gulf separates the languages, the process can be very difficult to accomplish. Programming language translation done "manually" has the same essential problems, but is usually less concerned about preserving any nuances in the original not directly affecting "performance". This can mean that programme translation sometimes requires not significantly less effort than programming anew (see Wolberg's 1981 report). Anyway, in both natural and programming language contexts, the prospect of automating the process beckons with siren-like attraction.

In natural language translation, though much work has been done and continues to be done, little progress has been made in applying such understanding very far. After the colossal effort in this direction over the last thirty years, the result is an ability to translate only very limited context material, with substantial preparation required before, and human finishing required afterwards. This is itself a useful ability, but the wider prospects seem dim. The problem is in understanding the very nature of language, and the way it is so tightly bound with the contexts of culture and experience. Winograd (1984) gives a general overview of the abilities and limitations, and earlier goes into great detail (1983, 1981) - a recent more technical overview is

done by Tucker (1984). The newer approaches attempt to build structures representing knowledge as networks and so approximating "context".

In translation between programming languages, the situation is complicated by the dual nature of programming languages: both human communication languages and formal computing models. Between formal computing models, translation is a quite tractable process, indeed including the essential and commonplace process of compilation. Again, most programming languages are as powerful computing models as we understand, and any such language may be translated to any other, by any other. In practical application, this approach has also been used to bridge from one language to another. In most cases, the bridge has been from older languages to newer languages perceived as "replacing" them: several very early languages were semi-automatically translated to Fortran, and before long Fortran to Algol (see Pullen, 1964, for example, with matching 'goto's). Fortran, Cobol, and Algol all to PL/I was especially common - Griffiths (1977) gives a short tutorial on the subject in general. More recently Fortran, Cobol, Pascal, and others have been considered for translation to Ada, Wallis (1985) surveys these efforts. Translation is also done from software literature rich languages to languages more commonly implemented in an environment of concern, C to Pascal, for example, in attempting to make Unix software more available for small personal computers (Carnevale, 1985). No review, however, seems very enthusiastic. The consensus seems to be that it would be very nice if it worked well, but in practice, it doesn't. At best, it is seen as an aid in text processing. The performance of such translators is similar to that of natural language translators: they seldom attempt to translate all of a language, they leave many capabilities untouched in the target language, and they require skilful "finishing" by a programmer . As such they are applied in some circumstances, most commonly in converting software that represents a sizable investment to a new language base. Unavailability, inefficiency, unreliability, lack of support: all are reasons for changing languages. However, the task is almost never trivial and can be a severe undertaking. Moreover, the results can be unsatisfying, and it does happen that even large programmes end up being entirely re-written "by hand". But there are cases for very useful application, and the factors include the size of the software, how suitable the software is for new applications anyway, and the "difference" between source and target languages. Between very similar languages results can be surprisingly good: in translation from Fortran to a structured fortran language, like Ratfor (Kernighan, 1975) or EFL (Extended Fortran Language, see Feldman, 1979), even "structuring" of control flow has proved surprisingly reasonable (Baker, 1977). Freak (1981) has used similar methods from Fortran to Pascal for conversion of numerical libraries.

The efficiency of programmes, often a prime concern, would be particular to the cases of individual languages. The particular linguistic structures involved might allow translation to very efficiently implemented language or not. In conversion of numerical software from Fortran to Algol 68S it is reported (Prudom and Hennell, 1977) that target programmes were often more efficient than source programmes! Because of the offerings of Ada for exceptions, generics, and packages, however, Wallis considers similar automatic conversion of portable numeric software to Ada not worthwhile.

Surprisingly, then, Albrecht et al. (1980), not only describe their Pascal to Ada translator, but also their Ada to Pascal generator! But the method is to restrict the Ada source to a Pascal sized subset, and to translate the Pascal to very Pascal-like Ada. And seeing that Pascal is the conceptual base of Ada anyway, the

the surprise diminishes. But is a useful tool, and allows reasonably easy integration of existing Pascal programmes into Ada systems, and allows Ada programmes, if appropriately written, to easily travel as Pascal. Such two way translation is quite uncommon.

Between natural languages, automatic translation might well assist the routine translation of industrial instruction manuals. Between programming languages automatic translation might well assist the translation of totally developed and unchanging linguisticly simple software. And while translation to a humanly readable programming language is the most desirable result, still sometimes useful might be translation to a form "unreadable" but computationally compatible with the target. So even a translator such as from Lisp to Fortran (offered, for example, by Systems Research Laboratories, 1985) might well prove a reasonable tool. The task of translating programming language is clearly more tractable than that in natural language. Translation of the formal computing element is, after the extensive research in this area, reasonably straightforward. Even that process may well yield programmes substantially less efficient in implementation, but not necessarily so. But it is the other element of programming language, the human face, that is so much more difficult to transfer yet preserve in translation. This is precisely the area where linguistic study in general finds it must go beyond any technical questions of procedure and come to terms with the breadth of human context. This is not simply a theoretical problem as described by Quine (1953 and 1960), but is a real problem in practical translation too (see Larson, 1984) - translation cannot be done without "knowing" meaning, and much falls within the scope of interest. In that aspect, as between natural languages, so between programming languages. That context of the application, and the paradigms of programming, are not part of the formal language, and so cannot be formally generated. However, the area for research in programming as well as natural language translation systems is explicit representation of that paradigm knowledge. Elliot and Holliday (1987) report on one of the first results of this approach. The translation is surprisingly good, but the source and target concerned, Fortran to Ada, is not sufficiently difficult to allow many conclusions to be drawn.

Automatic translation is possible in a limited way, though it cannot meet the requirements demanded in providing solutions to programming language diversity. Automatic translation can, with even a simple translation procedure, and a good text editor, greatly reduce the effort of translating a bulk of programmes between not too different languages (as proposed by Carnevale). To seriously challenge the diversity problem, however, it would need to be possible to easily translate most programmes between most languages preserving whatever is of linguistic value in each. The human factor in programming language makes matters very difficult. Wallis: "A change in language implies a change in the ways of thought suggested by the constructs of the language". If perseverance seems worthwhile, the newer approaches for natural language translation might be applied, using representations of knowledge about two languages to better effect a better mapping. In any case, considering the difficulty of this and the management problems that seem implicit, it doesn't seem appropriate to continuing use of differing languages.

SECTION II. INCLUSION
FAMILIES OF COMPATIBLE LANGUAGES

While compromise on minimality or multiplicity cannot be well defined and can only thus resolve problems of diversity in a practically limited way, strict inclusion of programming languages requires more rigour but can yield more flexibility. Inclusion is another approach to having multiple languages in cooperation - it's an approach with popular application and success. The effect of language inclusion is compatibility between languages, with the degree of compatibility varying with the degree of strictness of inclusion. The result is a limitation of diversity problems, and a way away in new language design. Working from an existing language to a new one, inclusion itself offers two alternatives: subset and superset.

Language Subsets

Language subsets are designed, developed and used for a variety of reasons. It may be that a language subset is designed and proposed to supplant (or usurp) an original: a drive to minimality. However, in designing a new language as a subset of an existing one the more common goal is cooperation with some other continuing language, the reasons most commonly concern implementation ease, pedagogy or very often both.

Fortran, for instance, has many many subsets for these reasons. Many "fortran" languages lack some data types ("double precision" or "complex" perhaps), some statements ("rewind" or "block data") or some input/output facilities - ease of implementation, ease of pedagogy. For much the same reasons, Pascal, another language widely used in education and limited use situations, is often subsetted - for the most part in minor ways: probably the most common subset might involve the non-implementation of procedures passed as parameters. The approach of Holt et al. (1977) in establishing a sequence of subsets ranging from the minimally usable to an entire language, is especially aimed at teaching. Students progress up through the subsets, yet at each step need only be concerned with the language so far learned. The approach is used in practice for teaching PL/I, and a similar method is proposed for teaching other large languages, Algol 68 (Pagan, 1980a), and Ada (Pagan, 1982). A more general approach is advocated by Bossi, Cocco, and Dulli (1982) in specifying a hierarchical decomposition of Ada for teaching. In this approach, all the smaller languages are sublanguages of Ada, but might themselves be partially disjoint.

But large and complex languages have been subsetted for implementation ease alone. In spite of large language promise, the diversity problem alone often seems to temper enthusiasm. Accordingly, language acceptance and language implementation often happens in steps, subset first. Algol 68 requires a large implementation effort, but acknowledged subsets allow reasonably economical implementation by leaving out facets that are difficult to implement - and thought less important in application. Algol 68S, for example, was specified and recognised very early for the important area of numerical applications (Hibbard, 1977). It might be wondered if the Algol 68 project might have had more popular success, had the inclusion progression been reversed, with the smaller language introduced and established, and the larger offered later as a "superlanguage".

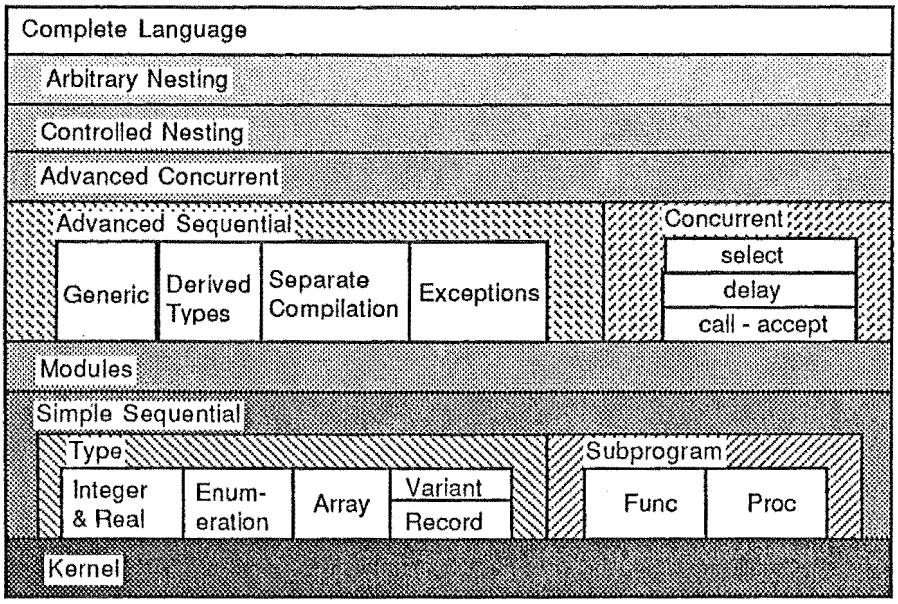

Figure 4.2.1: Bossi et al.'s decomposition of Ada into a language hierarchy. [Adapted from Bossi, Cocco, and Dulli, 1983]

Probably the most hotly debated language subset issue is that involving Ada. The issue arises because of the size and complexity of the language, and the many difficulties that implies, articulated most notably by Hoare (1981). The reasons and suggestions for a recognised subset have been detailed by Ledgard and Singer (1982), and a response made by one of the designers, Wichmann (1984). In arguing for a subset, Ledgard and Singer suggest that the benefits of a subset would be both in ease and in quality, and would affect implementation, teaching, programming, and system development - all in a positive way. Essentially the argument is for a smaller Ada, but in addition to the larger Ada rather than instead of it - accepting the larger Ada as a completed language design. It was also pointed out that the existence of a subset might better the popularity, and hence support, of the language by attracting those disgruntled by the size and complexity; this supported by the results from canvassing opinion (see Skelly, 1982). The idea of standardisation itself is still seen attractive, obviously, but of course really only so in accordance with the attraction to the final result. The argument against a subset is essentially one in favour of the design of the larger language, and that it is only appropriate to the requirements - no smaller language would meet them. The standard did specify a single language for embedded applications in the military context, one language to span very difficult and not well understood programming in a large context. Anyway, Wichmann argues, there is a problem with small languages too: they invite and lead to (often non-standard) extension. Actually, in practice, there seem few successful languages at all that have not been both reduced and extended. There have even been proposals to extend, even legally, Ada itself (with a macro-processor: Chambers, 1982), and the many Ada PDLs could also perhaps be seen as differing extensions.

And of course there is a central difficulty with Ada: on the one hand it is the product of the attentions of many many programming language experts around the world; on the other hand it is the property of one organisation, designed and acquired for a particular purpose. While subsets might well be of

use in some places in the computing community, that does not seem to be of concern of the proprietors. Avoiding the "proliferation" of languages was a design goal, and problems of incompatibility are feared. The problems of those hundreds of partially compatible, partially incompatible, "fortrans" must seem best nipped in the bud. In essence it is an argument against the need for diversity and thus against diversity, compromise or no; so of course not easily resolved by reason alone. Ada advocates continue to insist that the design process was one of reasoned consensus, and that after such a monumental effort divergence is now almost irrational (see Sammet, 1986, for example). Officially, the issue is now closed with a decision against any official subset, and it seems control will be tight over the language and its name. This attitude might be argued to be precisely that to spark subsets anyway, unofficial subsets, with all the mutual incompatibility that entails, but with care in use of the name.

Language Supersets

Language supersets have also been designed, developed and used for a variety of reasons. In some cases, extensions are made in order to improve some perceivedly weak point in an original language; in other cases a new language with different application emphasis is the goal. A superset is a means of saving work while retaining some possibly useful compatibility.

Language enlarging includes much practice involving well known and popular languages. Fortran (pre 1977), for example, has had many extensions, many made along similar lines though often differing in detail: common extensions are: free form input/output, a "CHARACTER" type, and some form of "structured" control. In the use of Fortran, so popular a language, some extensions themselves have become very widespread. For example Watfor (Shantz et al., 1967) and many similar products were specifically tailored for student programming and extensions were integrated into the language. Ratfor (Kernighan, 1975) and many similar products were specifically designed to make Fortran more palatable to users of more modern languages and extensions were preprocessed into Fortran itself. Such Fortran extensions have been sufficiently popular and numerous to make the progress genuine evolution of the language (see Meissner, 1975, or Haring and Schechtner, 1983 for surveys describing the evolution in detail). Fortran "77" (Wooley, 1977, Brainerd, 1978) chose between and united many extensions in design of a new widely sponsored extension language. Initially its impact was small, and the earlier "standard" still dominant. But as it is largely a superset, new requirements for Fortran implementation have favoured the new "standard" thereby catering to old and new, and so establishing the new. The unfortunate aspect about language supersets is that while all supersets retain compatibility with an original, supersets are usually not themselves mutually compatible. The multiplicity and diversity of language implementations make this a particularly bad problem for Fortran, as noted and catered to by pragmatic devices such as the PFort (Ryder, 1974) conformance verifier.

The preprocessor method of language increase frequently used with Fortran, especially before Fortran 77, is very common, probably the easiest method, and also gives rise to possibilities for cooperation using the common base. In this method the new language, Ratfor for instance, is implemented by "preprocessing" - translation - to Fortran itself. However, not all new facilities can be introduced this way,

and there is concern about efficiency in passing through more languages on the way to implementation. A hybrid method between language change itself and preprocessing is suggested by Zelkowitz and Lyle (1981), who discuss how an macro facility inside the language implementation level could be used to make language change easier and faster yet integrated in the one place.

Algol (pre 1968) too has seen many extensions, though an evolutionary path was not really ever sponsored as with Fortran. While Algol "W" (Wirth and Hoare, 1966), extending for records, dynamic data, and primitive input/output, was considered, it was a different choice that became Algol "68" (see Hoare's recounting in "The Emperor's Old Clothes", 1981). It would have been similar if PL/I had emerged, as once planned, named Fortran VI (Radin, 1978).

The second category of language superset is a much larger jump: an newly oriented language on an older base. In this way, many language fundamentals may be accepted without argument - attention is then more available for details of the new design components and attendant implementation. Probably the best known examples in practice involve the specialisation of general languages to new application areas. Simula, for example, was built on Algol but added much new, most notably the "class" concept (Dahl and Nygaard, 1966). Recently the C language, probably because of its role as the system programming language for Unix, has founded many supersets, most adding ideas from other languages. Stroustrup (1983), for instance, has added the "class" concept from Simula, so making the relationship between the new language and C similar to that between Simula and Algol. Cox (1983) discusses an object oriented C (OopC), (and later Objective C - Cox, 1986) with strong objective to "evolve" from C with influence from Smalltalk. Katzenelson (1983) introduces "enhanced" C, which encompasses a set orientation influenced by SetL (Harrison, 1973). Kilian (1985) describes a C with concurrency synchronisation added through "critical regions" proposed by Owicki and Gries. The variety is bewildering, but by keeping the base common, all such software is able to retain contact with software in C.

While it is not always clear whether some new direction is a specialisation of a general language, the approach should probably be carefully considered. In the Ada effort, for example, there was some investigation to see if the requirements might be fulfilled by building on an existing language so making a superset, or less formally as a conceptual base (see Whitaker's 1978 history). However, in that case only the last option was deemed worthy or pursuit. One price thus paid was the loss of potential compatibility of existing language implementations, programmes, and programmers. Naturally, availability of development resources will always influence choice in such matters.

Strict containment, inclusion, of languages clearly has strong advantages. Even where compromise is the theme, though, as perhaps the Ada spirit best shows, there are grave pitfalls. Firstly, few languages attain success or notoriety with a total absence of known blemishes, and it is annoying to perpetuate them either in subset or superset. Secondly, fragmentation along similar lines may even worsen diversity as subsets or supersets are not necessarily themselves contained in any strict way. Upward compatibility may be assured, sideways compatibility less so. And thirdly, the entire approach is at best a limitation to new diversity, and does not address resolution of old diversity. Many extant languages simply don't nest, and some notational initiatives are perhaps worthy of more freedom.

SECTION III. FLEXIBLE DEFINITION
LANGUAGES WITH BUILT-IN LEXICAL FLEXIBILITY

In seeking the compromise often necessary in establishing a language, differences in agreement on detail have sometimes lead to a degree of flexibility in the language surface. Where this is allowed, there is typically specified a lexical level expressed not in surface symbols to be actually used, but rather in parameters for surface symbols, parameters reflecting several choices for actual surface symbols. This flexible approach to language definition can thus be seen as another cooperative arrangement in tackling the diversity problem. It is just a lexical flexibility, but the division between lexical and syntactic concerns is not a precise one. Anyway, lexical issues can sometimes seem quite important.

In a simple way, the approach might be seen as stemming from the diversity of display equipment and the need for a language, particularly in more primitive times, to span several character sets or indeed display techniques. Beyond the question of availability, however, is the root question of desirability. The technique was first widely discussed in the debate leading to Algol 60. All languages span some community, but Algol, more than most languages, had as an objective to span communities of differing natural language and "natural culture". Perlis (1978) relates:

"The proposal that there be three representations for the language: reference, publication, and hardware was a master stroke. It simultaneously freed designers, implementors and users from pointless debates on issues over which none could expect to have much control. It made possible the appreciation of the ideas, the intent and the feel of the language, without the need of prior language alphabet projection onto local (usually limited) character sets. As with so many clarifying concepts, the 'three representation' proposal arose from a need for compromise. Joe Wegstein has reminded the author that, after two days of probing attitudes and suggestions, the meeting came to a complete deadlock with one European member pounding on the table and declaring 'No! I will never use a period for a decimal point'. Naturally the Americans considered the use of a comma as a decimal point to be beneath ridicule."

```
co List length, number of positives, negatives, and sum of numbers... co
        begin
        int num:=0, pos:=0, neg:=0, real absum:=0, x;
        while read(x); x ne 0
              do absum plusab
                    if    num  plusab 1;   x gt 0 \
                    then       pos plusab 1;      +x;
                    else neg   plusab 1;   -x;
                    fi
              od;
        print((num, pos, neg, absum))
        end
```

```
# List lenth, number of positives, negatives, and sum of numbers... #
(       int num:=0, pos:=0, neg:=0, real absum:=0, x;
        while read(x); x ≠ 0
        do absum +:= (num +:= 1; x >0 | pos +:= 1; +x; | neg +:= 1; x; )
        od;
        print((num, pos, neg, absum))
)
```

Figure 4.3.1: An Algol 68 program in two different program representations.
[Modified from Lindsey and van der Meulen, 1977]

In the definition of Algol 68, techniques were more sophisticated and involved several layers (see van Wijngaarden et al. 1976). The bulk of the definition concerns the "strict" language, with no terminals, in

the earlier sense, at all; only abstract names for symbols ("bold if symbol"), and those seldom directly were permitted. The strict language must in practice be represented, however, and there is choice in this "representation language". Offered as one such is the "reference language", and elsewhere described is another as the "hardware language", specifying precisely practice for use with coded character sets (Hansen and Boom, 1978).

Even within the strict language is some facility for choice, thus duly available in representation and so reference language too. For instance there is both the "bold if symbol" and the "brief if symbol", in the reference language "if" and "(" respectively. In a representation language itself there may be choice for one symbol in the strict language, for instance the strict language "is symbol" is available in the reference language as ":=:" and "is". Composition is allowed in symbol representation, and indeed so is duplication of representation, or of parts composing a representation - both subject to the disambiguation possible from the syntax of the strict language.

Probably the only wide application of flexible definition, however, is in coping with implementation environments without all the characters desirable being available. This is not uncommon because of the way that character sets have grown larger while older equipment and software continues to survive; because there are different character set traditions anyway; and just because there are idiosyncratic facilities everywhere, both in hardware and software. So, for example, PL/I in the IBM environment (IBM, 1976) provides the choice of 48 or 96 character sets suitable for the older BCD and newer Ebcdic character sets used in IBM systems. Pascal requires far less symbols than PL/I, but it still offers in Wirth's original report (1976) and in the new ISO (1980) definition the alternatives "(*" and "*)" for "{" and "}". In considering these levels of lexical simplicity, it might be remembered that there is some provision in the definitions of some standard character sets themselves. However, this is usually simply to match orthography of a particular context natural language and culture, and consists of substituting one character for another, or simply adding and subtracting characters (see MacKenzie, 1980, for further discussion). Character set definitions and especially encodings are seldom used as tools in language definition explicitly. Most languages consider spanning implementation bases, and character set provision is normally an aspect of those implementation bases, not of the languages themselves. Practice anyway would depend on the often widely varying nature of character sets, and there is a diversity problem there too. Further, character set flexibility is itself quite limited: it encompasses few symbols and few choices, and is generally insufficiently versatile for much interesting language flexibility.

The flexibility of an abstract lexical level is clearly useful: it is a immediate link tying together several otherwise divergent language variants. In practice, however, the flexibility is limited to the simple and immediate lexical situations as discussed, and there involving only simple and immediate lexical choice. While there is no theoretical limit to greater flexibility, the ensuing practical problems of determining compatibility would eventually prove a nightmare in scanning and parsing. Firstly, in order for diversity to be well thwarted, all implementations must know all variations, and know how to cope with them. For many implementations this overhead would undoubtably seem a great nuisance. Secondly, as the flexibility and its provision descend deeper into the language structure, that overhead grows beyond reasonable bounds. And a totally flexible frame is no frame at all.

SECTION IV. COMMON IMPLEMENTATION
COMMUNICATING BETWEEN PROGRAMME COMPONENTS IN DIFFERENT LANGUAGES

In tackling practical problems in programming language diversity directly, easily the most common tactic employed involves reliance on differing languages possessing a common component in implementation. While the establishment of a programming language is expensive, the establishment of the more primitive basis of the implementation is often even more expensive. In practical computing, a tunnel from one programming language island to another is a common expedient. There are several methods that have been employed, several layers of implementation environment that have been tunnelled through, and usually with much success in tackling many specific situations.

Low Level Bases

Certainly the earliest and most widespread form of this approach involves the machine level, the instructions of a specific computer. This level is the deepest that programming language implementors normally delve in creating a usable language, and so is a first common platform for any language implemented. In tunnelling between languages, a programmer must understand in at least some ways how each language is implemented, and contrive to make use of any common methods in uniting the two. This process ranges widely in difficulty according to the method of language implementation and the underlying model employed. Where, for example, implementation of languages involved is by translation - compilation - of language code to "machine" code, machine code resulting from differing languages can often be combined and used together.

Assuming sufficient skill and adequate tools the process is possible in general, but common features for external compilation of programme components, common higher control structures (procedures and blocks), and common data formats sometimes make some specific combinations quite straightforward. Within an environment where this method works well, it often works very well: it is both direct and efficient. Some machine and operating systems are sufficiently numerous and some programming language implementation methods sufficiently common that such cooperation can long continue. Where the link from one language specifies and assumes the other, some specific tailoring can be made, still making assumptions about implementation, but tailoring the interface better on the basis of whatever knowledge of the implementation is common between source and target environment. For instance, procedures in IBM PL/I may be declared as "Fortran" or "Cobol", and upon the assumption that those languages are those commonly available in the environment that PL/I is itself available, connections can be made.

This has been suggested more generally as something to be tailored to individual environments, and for instance such subroutine based "mixed language programming" has been arranged using Fortran and Pop-2 (MacCallum and Schafe, 1974), and using Fortran and Pascal (Mohilner, 1977). Where the supporting environment provides dynamic access to subroutines, and where the environment permits internal manipulation to maintain integrity, this can also allow more flexibility. In particular, it can become possible to call from an interpreter implementation of a language to subroutines compiled to the common base

from another language. In this way, special environments can support Prolog interpreted programmes calling Modula procedures (see Muller, 1986), or C functions (see Matthews, 1987).

Subsequent discussion has lead to more direct advocacy for general "mixed language programming". Einarsson and Gentleman (1984) are particularly interested in the problems of availability of numerical subroutines to other languages, but do relate that problem to other applications that might span language competences. They suggest a common subroutine interface standard for mixed language programming, with access to the facility in various languages, each language providing enough cues for building of mixed language programmes to be possible automatically. In such a cooperative system, even data typing could be possible, assuming enough common background, and an earlier proposal by Darondeau, Le Guernic and Raynal (1981) is suggested to import and export types with data. As Einarsson and Gentleman conclude: "The main part of the realization is achieved by standardising interfaces and requiring operating systems to permit the usage of mixed languages." In the particular area of interest, these might conceivably be possible, but hardly trivial, for such a standard to be established.

Vouk (1984) is particularly interested in mixed language programming for avoiding manual conversion of large programmes from language to language. Using experience from such conversion and related software engineering practices, he argues that the approach can be cost-effective because of the difficulty of automatic translation, and the continuing high human and management costs of less automatic methods. Vouk also sees standardisation as the key to effective mixed language programming, and seems optimistic about the possibilities for automatic programme binding suggested by Einarsson and Gentleman.

Wulf called for standards in this area as early as 1972, but was doubtful then how both language and implementation diversities could be sufficiently bridged. There has been some effort approaching such standardisation by Ansi (1982) since, but the guidelines are vague and not nearly precise enough to support an automatic system. In a general context this is not too surprising, as the attraction of new formats and new structures at the application level is obviously still great in encouraging diversity to continue.

In wider scope the problems are even larger. Some programming languages and some environments are really quite unsuitable, and the process is then difficult and largely unreliable. Firstly, languages (or implementations) without conception of separately translated components present difficulties; languages without any well developed concept of separate sub-units present even more difficulties. Compiled language implementations suggest the possibility of actual manipulation of the machine code after translation and before execution (!) - but even so, interpreted languages are much less accessible and manipulable. Secondly, even when connection and communication are possible, discovery of the exact protocols can still be an ordeal. The subject is often poorly documented to the point of secrecy: it's simply not normally understood to be appropriate to programming language use. Thirdly, even when conditions are appropriate and a path hacked out, the path is likely to be notoriously unstable. Often workings at the levels involved are more subtle than are imagined by programming language users, and connections may fail in circumstances unforeseen. More alarming, though, is that as the implementation and connections of a programming language are regarded as below programming language users; they are subject to change without notice in maintenance of the support level programming. So the path is very precarious. Fourthly, even when all the

above problems can indeed be borne, the method is still not portable. When the language implementation changes, the connection is gone; when the system environment changes, the connection is gone. In practical everyday programming, both really do happen.

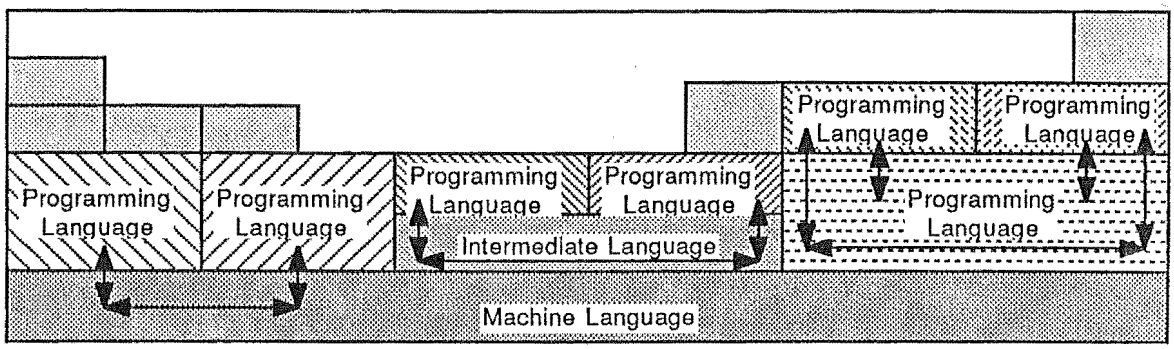

Figure 4.4.1: Communication between common levels of programming language implementation.

Intermediate Level Bases

Many of the problems involved in using machine environments as common bases for language cooperation are similar to problems involved in programming language portability. In tackling the diversity problem, one approach proven successful in tackling the portability problem seems of interest: the use of an intermediate language in translation.

This method has been in consideration and use for quite some time, and has seen substantial success. In employing the method, a programming language is normally translated to the intermediate code, then the intermediate code is normally translated into some working model, thus effecting implementation. In this way, the "depth" of implementation effort is limited to the intermediate level, and so in tunneling between languages so implemented, only the intermediate language need be involved. So doing reduces or eliminates some of the problems stated above: it yields reliability and portability. If the intermediate language is well designed and well defined and documented, the other problems are also reduced. Brown (1972) shows the different ways that an intermediate level can be used, and Newy, Poole, and Waite (1972) review and evaluate the approach in general.

It has long been seen that a single intermediate language for implementation of many different languages has great advantages in reducing programming language implementation effort and aiding portability. As early as 1958, for example, suggestions were made by Strong et al., explicit proposals by Conway immediately followed, and there was much ensuing discussion. In the interim surprisingly little has actually been accomplished along this line. An interesting early proposal by Landin (1966), suggested using a mathematically powerful functional language, Iswim ("If You See What I Mean"), as a base, with other languages implemented portably one level up. But concern for efficiency has often been held more important than concern for portability, and any use of intermediate code will reduce efficiency - intermediate code not specific to one language will reduce it further. And while one advantage of use of an intermediate code is that

it can be designed to be easier to work with in generating code, this too will be reduced by requiring generality over languages. Accordingly, most intermediate language success stories, Snobol, BCPL, and Pascal, say, have been with intermediate languages designed, if not totally specifically, then at least with a particular language in mind. And recently, totally specific: the Ada intermediate form Diana (Goos, Wulf, Evans, and Butler, 1983) is specifically very Ada oriented in intention to be useful across several Ada tools - "formatters (pretty printers), language-oriented editors, cross-reference generators, test-case generators, and the like". Some Pascal intermediate codes have been used for other languages, notably the UCSD Pascal P-Codes (Clark and Koehler, 1982) in implementing Modula and other languages. Specifically "combined" systems also sometimes have a common intermediate level for their language components. Poplog, for example, provides connection between Pop-11, Prolog, and Lisp at the level of "Poplog virtual machine code" (Hardy, 1984). So in that particular context there is the possibility to choose the best component of the three for particular parts of application programmes. These approaches do allow communication through the intermediate level, but span only a limited language set.

However, there has been development of more general intermediate strategies for portability. In the Stage2 system (Waite, 1970b), the intermediate basis is a macro processor, and in the descendent Janus system (Coleman, Poole, and Waite, 1974), the intermediate basis is a more general language. Both systems are called "mobile", rather than just portable, because of the importance attached to easing the establishment of the system in new environments (see Waite, 1970a). Such systems seem to offer great advantages in portability, and indeed promise to fulfil or better the dreams of 1958. Neither Stage2 nor particularly Janus has fulfilled such promise in practice, probably because they again present yet another standard to be adopted - particularly with the more specifically tailored orientation of Janus. And there is still the problem coping with the choice between efficiency and portability. In a more recent attempt to address that difficulty, the Amsterdam Compiler Kit (Tanenbaum et al., 1983) does involve an intermediate language, EM, but makes extensive use of portable and non-portable code improvement strategies (see Tanenbaum, van Staveren, and Stevenson, 1982, for example). The EM system should offer a good opportunity for communicating through an intermediate common implementation spanning many languages.

All these systems are operational, yet to become really successful they really need to become both widespread and widely used. At this point the difficulties with efficiency and ease of implementation must be remembered, and even beyond these there is a diversity problem with intermediate bases. Cooperation between languages by reliance on common intermediate bases would become less useful if it were possible only in partitions. In particular instances, of course, useful it still could be.

But there remain problems. Again, not all languages could be well adapted to the approach. Again, not all programmers will wish to delve so deep. And while less so than with a more basic level, an intermediate language may still be less accessible and still more liable to swaying change than would be desirable. Moreover, while any such common multiple base intermediate language may well advance portability, not all language implementations will use it. Strong factors still, efficiency and secrecy may well put many programmers off the approach altogether.

Higher Level Bases

Sometimes, the base upon which a language is built is a level higher still. In building one programming language, it is sometimes done to build on top of another. This is normally achieved by interpretative implementation in a general purpose programming language, or by translation by, to, and thus through, a general purpose programming language. Again, the intention is often portability and ease of implementation, and again this approach may prove useful in linking programming language implementations. Where two languages are themselves implemented by means of a common third, the depth of implementation is that common third. As a means to portability and implementation ease, the approach has been popular and successful. Perhaps the most frequent building block has been Fortran, and it presents many of the advantages to newer programming languages that an even longer dead natural language, Latin, is held to present to modern European natural languages. It's very widely know, is very established, and is not still rapidly evolving. Other general languages have been employed too, Cobol, PL/I, and Pascal for instance - clearly a popular language increases the portability effect, clearly a well supported language increases the ease of implementation effort.

There is wide variety in the application of this approach. The simulation "language" Gasp-IV (Pritsker, 1974) is in fact a set of Fortran subroutines and so not only permits but really requires access to the Fortran base. Even a more separate language like GPSS normally assumes a Fortran base and so encompasses facilities for access to Fortran subroutines, (see Bobillier, Kahan, and Probst, 1976, and Gordon's 1978 history). SimPL/I (IBM, 1972b), to stay with simulation, was conceived and totally designed to be translated into PL/I. The symbolic mathematics language Formac (Sammet and Bond, 1964, IBM, 1965, 1967) too was implemented as a preprocessor for Fortran, and then later for PL/I. In fact this is the way to specialising languages in several fields: Ng and Marsland (1978), for instance, discuss using preprocessing to introduce graphics facilities into several different languages - there are many similar such examples.

Ratfor (Kernighan, 1975) is necessarily a preprocessor for Fortran to allow "structured programming" while maintaining strict compatibility with Fortran. In fact many such Fortran preprocessors have been popular as Fortran became less acceptable yet reliance on it remained (typically because either other languages were not implemented in the environment, or compatibility with older and continuing software was essential). So this strategy was a lot about cooperation, though usually strictly pragmatic. In fact the approach was often seen as a possible route for that great quest, doing away with Fortran altogether (see Fraley, 1977, for example, or Arisawa and Iuchi, 1979). At least with regard to simple interpretations of quality, portability, and cooperating in diversity, the approach has been quite successful. Kernighan and Plauger's (1975) Ratfor based "Software Tools" was about teaching and learning, but with directly useful programmes. These programmes, similar to some Unix facilities, have since been harmoniously established in many many different support environments, suggesting the possibility, detailed by Hall, Scherrer, and Sventek (1980), of a "virtual operating system" at the higher than programming language level. Extending the idea, Snow (1978) discusses translating the programmes to BCPL before implementation. (Interestingly, several of the programmes began in BCPL, and evolved their way with language evolution through B, C, and Ratfor, before returning.) But translation via another programming language is not without extra costs,

and even in this relatively simple translation, those who take the efficiency of their Fortran programmes seriously may find the costs of preprocessing unacceptable (see Meeson and Pyster, 1979).

But the approach is not restricted to attempts to hide the relics of long ago. A suggestion for extension of Ada (!) that avoids legal problems about the name as well as technical incompatibilities also uses a preprocessor to effect the larger language (Chambers, 1982). Some of the Ada PDLs are almost formal enough and completely implemented to be considered programming languages anyway, and incompatible in diversity themselves are only able to communicate through their common base, Ada itself. The Unix language tools, the parser generator Yacc (Johnson, 1975) and the lexical analyser generator Lex (Lesk, 1975), while both written in C do themselves generate programme components in either C or Ratfor (Johnson and Lesk, 1978). "Fourth Generation Languages" might also be included the approach. Burroughs' Linc (1983), for example, implements specifications of commercial transaction terminal systems by generation of Cobol programmes.

Probably the only common use of the possibility for connecting language gaps, however, is that concerning the between the new language and its host. In this way, GPSS when implemented in Fortran, and Fortran itself are usable together (though not trivially). In fact, if machine code is regarded as a programming language, this is how much system programming is effected. Wider use, though, communication between two languages implemented in a third, while possible, is probably rare, if only because occurrence of compatible neighbours is rare.

The approach of communicating through a common implementation in a programming language has the advantage of portability and (when used well) relative reliability for either language. It also has the advantage that the common element is better documented, and less liable to change, as with Latin. However, the implementation and the union might be clumsy and probably inefficient. Even more importantly though, as the implementation method is not so popular, is that a "tunneling" approach would for great success require widespread agreement on a common base. There is no universal high level assembler. The likelihood there ever shall be is probably diminishing.

SECTION V. PROGRAMME COORDINATION
COMMUNICATING BETWEEN PROGRAMMES IN DIFFERENT LANGUAGES

Direct connection between programming languages has seldom been possible within conventional language and operating system design. Indirect and more limited facilities, however, have been quite common. In supporting a general programming environment, operating systems typically include input/output facilities common to many or all programming languages. In this way, operating system and other service programmes may span several programming languages, and so also provide a means for connecting those languages. This method of employing different languages together is a very useful method and is widely employed in practical programming, but has historically received little attention in research.

Pre- and Post- Processing

Historically, this approach has been limited by the orientation of computing to a single programming environment. Initially, moreover, the separation of programmes in implementation typically leads to only the most primitive programme control, typically a simple list of programmes in order was well provided for. Accordingly, components in differing language environments were strictly sequential in time. When graced with a name at all, this approach is usually referred to as preprocessing or postprocessing, depending on which component is seen as central and which auxiliary. Little notice is taken of this method in the literature, yet the method offers much in circumstances where the sequential or "batch" conditions are not bothersome. In practical analysis of experimental data, for instance, the close bitwise control of a low-level language, BCPL say, might be appropriate for a first pass, manipulation in a more numerate language, Fortran say, might form a second. In seeking statistical information from a body of survey data, a first pass with SPSS (Nie et al., 1975) might generate much information in surplus, a second pass with a text editor might select the relevant component, and a third pass with a report generator might present or summarise specific findings. [This approach was used extensively, though not formally documented, in work I was involved with at Systems Dimensions Limited for the Government of Canada.]

Control Languages

While general purpose computing environments have long provided for simple sequential controlling sequences of different programmes, in possibly different languages, many environments now provide a more general control facility. Even earlier primitive environments had a tolerance for devious measures to enable greater control, via input/output to a control stream for example, but such practices were usually both subject to undependable side-effects and also quite inefficient. The new facilities have in fact been provided even in the contexts of some "off-line", "batch", or card-oriented environments, though they have flourished in the context of "on-line", "timesharing", or teletypewriter-oriented environments. The purpose of these newer control facilities remains as it was for even primitive sequencing facilities: to enable building a set of programmes into a larger organised unit. In their 1972 survey, Barron and Jackson use the development of programming languages as background to looking at "control" languages. They decide the dominant such language to that time, IBM/360 Job Control Language (1964b), resembles assembler, and the

languages then currently new resembled the earliest automatic coding systems. They argue that a control language is just a programming language for a particular purpose, and propose more be done to advance design. However, the problem with designing portable control languages seems to be the traditional close connection of them with operating systems, and the diversity in operating system design.

```
//C          EXEC  PGM=IEMAA
//SYSPRINT   DD    SYSOUT=A,DCB=(BUFNO=50,OPTCD=C)
//SYSLIN     DD    UNIT=2400,DISP=(,PASS),DCB=(RECFM=FB,BLKSIZE=400,LRECL=80)
//LKED       EXEC  PGM=IEWL,COND=(9,LT,C)
//SYSPRINT   DD    SYSOUT=A
//SYSLIB     DD    DSNAME=SYS1.PL1LIB,DISP=OLD
//          DD    DSNAME=SYSRI.PL1LOAD,DISP=OLD
//SYSLMOD    DD    DSNAME=&B(B),DISP=(,PASS),UNIT=2311,SPACE=(CYL,(5,1,1))
//SYSLIN     DD    DSNAME=*.C.SYSLIN,DISP=(OLD,DELETE)
//          DD    DDNAME=SYSIN
//SYSUT1     DD    UNIT=(2311,SEP=SYSLIB),SPACE=(1024,(200,40))
//GO         EXEC  PGM=*.LKED.SYSLMOD,COND=((9,LT,C),(5,LKED))
//SYSPRINT   DD    SYSOUT=A
```

Figure 4.5.1: Fragment of Job Control Language JCL/360, showing a sequence of programmes, executed on conditions (COND=) involving return codes of earlier programmes - this is the only control structure available.
[Text of "Catalogued Procedure" for IBM OS/360 PI/I (F)]

In attempts to establish machine and operating system independent control languages, the restrictions of some operating system designs exert limiting influence. So such "general" languages as described by Dakin (1975), Rayner (1975), Parsons (1975), Madsen (1979) or Moore (1979), for example, while no doubt very useful in what they offer for portability, are still unfortunately simple in the possibilities for coordination they can offer. Design for portable programming languages in general must be able to make many assumptions about the ability of some machine, even if it results in implementations varying in size and speed, as long as they are within some reasonable realms.

```
programme ; programme        # simple sequence
programme `programme`        # programme output as programme argument
programme &                  # programme in background
programme | programme        # programme output to programme input in parallel

if programme                 for name                 for name in list
then programmes              do programmes            do programmes
else programmes              done                     done
fi

case string in               while programme          until programme
string ) programmes ;;       do programmes            do programmes
string ) programmes          done                     done
esac
```

Figure 4.5.2: Control Structures of the Unix (Bourne) Shell.
(Conditional constructs use programme return values.)

But where machines differ, such design is obviously greatly impeded. No matter the equivalence of formal computing models, the pragmatics of machine ability are still important: size and speed remain great concerns. In the case of control programming languages the ability to structure programmes seems the main objective, and experience in structuring within other programming languages indicates that can encompass much: not only sequence, selection, repetition, but also recursion, parallelism, object orientation, and

arbitrary combinations. So for portability to be assured, languages must offer only a little, or supporting environments a lot. Some modern facilities might be regarded as bordering on offering sufficient facilities at the cost of portability by encompassing at least some non-trivial sequence control and at least some data control. Probably the best known and most influential such a sophisticated facility now is the Unix "Shell" (Bourne, 1978).

As an approach to challenging programming language diversity, there have been several limiting problems with coordination systems. Sequence control is now often highly developed, and similar to the sophistication of modern general purpose programming languages. Data control within the control language itself is also often sophisticated, and a wide range of data types and operations is not uncommon. However, between programming languages, communication is typically poor. From control language to application language direct communication is typically limited to a character string as an "argument". From application language to control language, direct communication is typically limited to a simple "return code" nominally for error indication. Even from application language to application language, the underlying an all-important common input/output system can often present less compatibility then may seem desirable. Many of the attractions of the approach were visible in OS/360 (for instance) twenty years ago; some of the limitations are visible in Unix (for instance) now. In the Unix pipeline and filter approach, processes may run in (pseudo) parallel, but may still be connected simply in a linear and unidirectional sequence. There are definite advantages of the parallel pipeline over a linear sequence in interactive use, of course, but more could be desired. Moreover, too specific a programming language dependence on a particular control language illustrates that a control language is yet another programming language, and diversity may yet increase on that avenue. There is a danger of a control language diversity problem, and contexts with multiple control languages must be careful to avoid consequent incompatibilities. Some versions of Unix, for instance, support several "shells", commonly both the "Bourne" shell, and the "C" shell (Joy, 1983). Relying on the command language alone to solve cooperation problems does not seem wise.

Language

In a proposal for "Modularity of Computer Languages" Beech (1982) suggests a generalisation of the control language approach. In essence, he suggests a hierarchical structure for programme coordination, where the execution of one programme may entail the execution of another programme, and so on recursively, with information passed in "call" and "return". A similar idea was advanced earlier by Jones et al. (1979), in suggesting there was little difference anyway between the notion of "programme" and "subprogramme". Separating the control language from the mechanism of coordination seems an important step; a control language is just another language after all. Another important point made by Beech is that with such separation the syntactic and semantic furnishing of coordination can be suited individually to different languages. In this way, the difficulty of linguistic evaluation can be somewhat avoided. But a difficulty does remain: is the coordination structure sufficiently general?

More general coordination is often available in environments supporting parallelism and message passing, usually for real time or control applications. Especially where such structure is available at the programme level, the flexibility might prove useful. Many environments for concurrent programming are

themselves language based (see Andrews and Schneider, 1983), and so are not directly suitable. However, other approaches include the required facilities with the general services of the operatings system. In Tripos (Richards et al., 1979) or Thoth (Cheriton et al., 1979), for example, programmes can be established separately but in (pseudo) parallel, and with sufficient knowledge of names can pass messages back and forth. Real-time execution efficiency is usually a primary concern of such systems, so programming languages leading to efficient implementation are usually used, and the system oriented about them. Tripos, for example, is centred about BCPL, and Thoth about the Thoth base language, Zed.

However, in the Jade environment for developing distributed software (Witten et al., 1983, Unger et al., 1984) the potential for such programmes to be derived from differing programming languages is explicitly noted. And indeed access to the communication protocol, JipC (based on the primitives of Thoth), is offered through several languages: C, Lisp, Simula, Ada, and Prolog (see Neal et al., 1984, and Project Jade, 1984). This does enable use of such differing languages together in the design of a single system. The need for execution efficiency in real-time systems is still compelling, and some languages might not lead to such efficiency as much as others. In systems development, however, other languages might have a role to play in prototyping, a main interest in the design of Jade, and interest in efficiency might focus on higher levels (see Lomow and Unger, 1984).

[A proprietary project with similar (though more primitive) goals to Jade, and also using Unix for a development base, was Code, a common development environment for real-time applications (Slekys et al., 1979). Where the key protocol of Jade is from Thoth, the key protocol of Code was from Mascot (Simpson and Jackson, 1979, UK Ministry of Defence, 1979), a more semaphore and queue based approach to synchronisation. Personal familiarity and involvement with Code is a probable influence in this general study.]

In Jade, all programmes, regardless of language, communicate using the same JipC protocol. If the language of one component programme were to be changed, therefore, no other component need be affected. In the access provided through the various languages, though, the protocol is explicit. In the Jade approach, this is perhaps only reasonable, as it is the JipC protocol that is the connecting link in both the development and target environments. But in a wider scope, there is a linguistic difficulty: JipC may not always seem appropriate to the level of discourse of many languages. Jade was not really concerned with the linguistic side after all, but was designed as a specific system building strategy one level up from, and so more portable than, the level of the operating system.

In offering a way to cooperate between language difficulties, the provision of facilities for coordination between individual programmes in differing languages seems very promising. Even in very restricted forms it can offer useful linguistic escape and recourse. As such coordination contexts have developed, the emphasis has moved away from specific languages charged with the coordination responsibility, moving toward command facilities shared by all programming languages - indeed all programmes - in a computing environment. While there are restrictions in present application, the reasons for their continuance do not seem necessarily strong, and an examination of a general capability seems indicated.

SECTION VI. CONCLUSION

In surveying cooperative approaches to the problems of diversity, it seems that this perhaps initially pessimistic view does in fact holds reason yet for optimism. While no practical technique discussed that is without some difficulty in spanning programming languages, several are, within bounds, surprisingly successful in practice.

	Advantages	*Disadvantages*
Translation:	• Good for one-time language moves. • Reasonable between very similar languages.	• Seldom preserves sytle. • Needs human finishing. • Very poor between very different languages.
Inclusion:	• Smaller languages with passage out - pedagogy, easy implementation. • Larger with passage in - extra features.	• Limited scope in compatbilities possible. • Languages compatible with a base language not themselves compatible.
Flexible Definition:	• Easy resolution of simple concerns usually involving equipment or orthography.	• Limited to lexical levels. • Difficult to implement and maintain compatibities if choice too wide.
Common Implementation:	• Generally applicable to many compiled languages. • Can be very efficient.	• Usually not portable or robust. • Difficult to use with interpretive or sophisticated implementations.
Programme Coordination:	• Generally applicable to almost all languages. • Reasonably portable.	• Flexibility limited by environment. • Inefficient transfer between language components.

Figure 4.6.1: Summary of particular advantages and disadvantages of individual methods of cooperation between programming languages.

As with methods of compromise, each method of cooperation does have particular advantages and disadvantages that will be important in specific circumstances. And also as with compromise, there are some aspects common to all cooperative methods. Most importantly, acceptance of several languages in cooperation makes the assumption that each language will be available in different environments. In the past this argument would have been sufficiently strong to throw doubt on the whole approach; it is still a concern. However, the subject of this exploration is the step beyond the "portability" concern; and even in practice the success of the portability pursuit - and of pragmatic success in environment standardisation - make software availability less and less a concern.

The practical strategies employed that offer attack on programming language diversity are not really deeply split by the distinction between compromise and cooperation. Languages are very general systems, and often the distinction is simply one of viewpoint in considering language interplay. As much as can be seen, however, the way to best coping with programming language diversity lies not in hoping to eliminate that diversity itself in compromise, but in learning to accept diversity in cooperation.

This distinction seems of critical importance: there is no need for compromise.

PART THREE

COLLABORATING

From INTRODUCTION:

Over all, this exploration of solutions to the problems of programming language diversity is structured with three divisions progressively narrowing in scope.

First is a consideration of the possibilities for reasoning to convergence in programming languages, thereby eliminating the problems of diversity by eliminating the diversity: integration at the roots of diversity. Central here is the question of programming language "quality" and how it might be evaluated. These questions are extraordinarily deep, and conclusions can really only underline the difficulty and importance of the problem. In these discussions scope is generally restricted to matters directly involving computing. As matters seem to lead away to other experience and more general enquiry, this is somewhat arbitrary and perhaps rather unsatisfying. Within the practical context, however, simply establishing the direction of difficulties is sufficient.

Second is a survey of established computing practices and techniques that in some way approach solution to the diversity problems. This a reminder that much that is impossible to totally resolve can still managed with tolerably. The approaches examined are grouped into those seeking "compromise": reducing different languages to one, and those seeking "cooperation": ways of using different languages together. Often the diversity problem has been only a minor or unstated concern of formal developments and studies, so some conclusions can be only informally drawn. Many strategies still have definite advantages in certain circumstances, however, and several deserve to continue being applied. But most strategies also have significant drawbacks that do keep their usefulness limited. Even so, some cooperative practices appear on examination to suggest further development of a more general strategy may be worthwhile.

Third is the proposal for a new general strategy for addressing and tackling the diversity problem. The philosophical problems realised in the first part of the study and the practical directions suggested by the second section do not quite seem in collision. Indeed it seems possible to practically evade problems of diversity as much as any particular programming language design permits, while avoiding any necessary implication of impossible universality. So such success must necessarily be limited, but some increased success is possible. The approach involves minimally collaborative design in programming languages, and necessary support from environments exterior to them. Though grounded in applicable practice, these proposals do not follow through in requiring specific action - there remains much choice. These are design principles for languages and support systems, and could be evolved toward. However, several programming languages are explored, and opportunities for useful access described or modifications suggested. To practically examine the question of the exterior support, two operating systems are also discussed, and the requirements of necessary design established. The importance of new software both to assist and take advantage of this "programming between programming languages" is illustrated with some new programming tools.

So more exploration and experience are indicated, and detailed study for particular programming languages and operating systems design would then be of most concern, as well as more attention to particular questions of performance and efficiency. Such particular work is outside the current scope, as indeed are several other directions suggested for future investigation. Explorations in representation diversity in other fields may deserve investigation next, and the balance of subject matter could be important as well as enticing. The contributions of this study are the mapping out of the vast area of this important subject, the review of relevant past practice, the development of a general strategy for progress, and the directions for its practical pursuit.

CHAPTER V
COLLABORATION
COOPERATION BY DESIGN

The starting point of this study was the claim that programming language diversity presents problems both bothersome and ongoing. Looking at the possibilities for reasoning to convergence resulted in deciding there was no sufficiently clear path evident now, and not likely to be one soon. Examining methods that have addressed simply coping with the problems showed that some specific situations could be well tackled. But more generally useful approaches were few. Compromise is difficult to establish anywhere, very difficult to establish everywhere, and anywhere and everywhere is too easily eroded. Cooperation seems the most attractive strategy discussed: the flexibility it offers seems appropriate to the slippery nature of diversity. In cooperating fully between programming languages, no programming language could demand reliance, and any programming language could allow escape.

Full cooperation, however, is not immediately achievable. In applying any particular strategy there are limits to the cooperation possible, or to the circumstances in which it is possible. These limits are imposed in part by the design of programming languages themselves, and even more by the design of the environments supporting them. The next step, then, is to see how far these limits can extended with appropriate design. That next step is to extend from cooperation to collaboration. Can there be - under what conditions and with what restrictions - programming between programming languages?

SECTION I. INTERPROGRAMMING: AN APPROACH TO COLLABORATION

In designing for collaboration, the experience from cooperation provides clues showing where to begin. Translation is useful when one language is abandoned for another, though only if they are not too dissimilar; even so it would be tedious where differing languages cooperate but remain vital. Translation shows the need for both wide generality and for immediacy of effect. Inclusion and Flexible Definition are both usefully immediate within close language groups, but not applicable to the larger span of diversity in question: again more generality is needed. Common Implementation is quite immediate, but again even more generality is needed than that possible assuming a common base structure. Moreover, even where the common implementation may encompass diverse languages, the integrity of cooperating programmes can seldom be assured. Programme coordination is by its "extra-language" nature usually safe from problems with integrity. However, the generality of the approach is subject to some provisions in the languages, and the flexibility with which it is applied is subject to some provisions in the supporting environment. But it is immediate - if slightly less efficient than if a common base were assumed - and potentially portable. So, in design or selection of a general approach to collaboration, direction must seek: wide generality of application; usefully efficient immediacy in use; assured integrity in application; and flexibility in configuration.

These requirements point not a need for magical decipherment, but rather to a straightforward approach of construction and evolution. Collaboration between programming languages can be made a

possibility - with sufficient generality, immediacy, integrity, and flexibility - by requiring within programming languages explicit facilities to suitably open their own context, and by requiring within supporting environments suitable facilities for communication and connection between those contexts. By requiring this small commitment by any language, and this slightly larger commitment by support environments, so all languages may be spanned. As the approach to facilitate traversal of diverse computer networks is called "internetworking", so this approach to facilitate traversal of diverse programming languages will be referred to as "interprogramming".

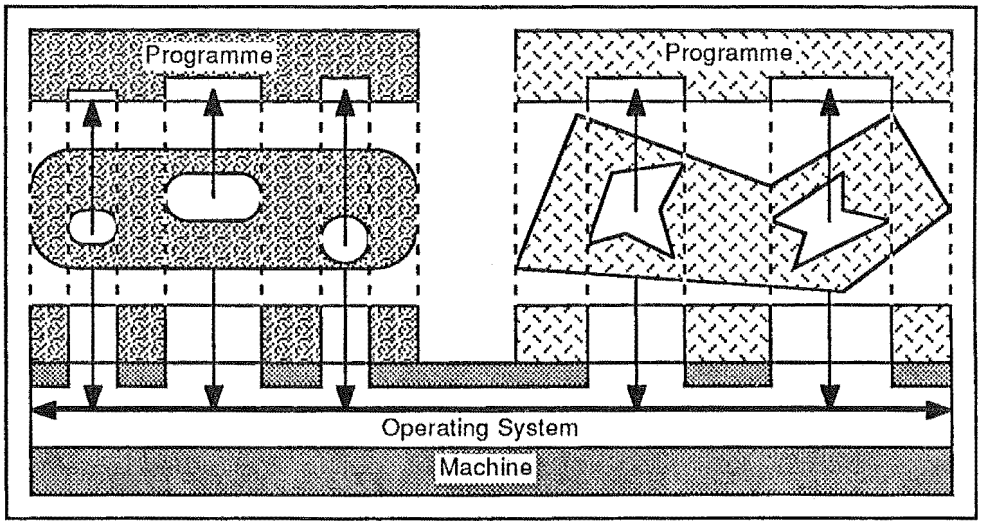

Figure 5.1.1: Interprogramming: communicating between programmes through "openings" in language contexts, the openings tailored to fit each language, but sufficient to communicate (possible indirectly) with any other.

The approach is a minimal strategy for directly satisfying the goals of collaboration. Other strategies could be built upon this - an important recommendation. The context "opening" of a programming language involves the acknowledgement from inside the language about the outside of the language. The acknowledgement should incorporate whatever communication and coordination is appropriate to the design. Input and output are simple context openings in most programming languages, and with careful design can themselves be quite suitable for direct communication; other openings can be more sophisticated. While it is important that languages allow programmes to share in whatever way designed, it also an important consideration to integrate the opening design with the language context itself. Only by allowing and encouraging this holistic orientation can the approach really work with, and not against, individual - and potentially individual - languages.

The communication and the connection facilities in the support environment might both be small and simple, but must involve the lowest level of discourse to ensure no language is excluded. The facilities must really be an integral part of the environment at a fundamental level to achieve reasonable immediacy and integrity. Neither communication nor connection facility should prove difficult within a modern practical computing environment. The facility might well fit in with existing design, might even be catered for already - most message passing and memory protection arrangements would be adequate.

While it would be new to pursue such a strategy explicitly, there are within interprogramming elements of many approaches to coping with diversity. Not only of cooperation, as might be expected, but of compromise too. As cooperation, the approach is clearly an outgrowth of programme coordination, for a general version of that is a requisite for support. And viewing that support as an at least partial level of implementation, the commonality required illustrates the similarity with methods relying on common implementation. This in fact takes to the point of union the cooperative strategies involving common implementation and programme coordination. And in the detail of communication there must be some provision for translation between contexts, not of the entire language, but in some perhaps component parts thereof: data, for example. As compromise, standardisation will clearly be useful to reduce the need for such data translation; more generally the whole strategy of regarding programming language environments uniformly might be seen as standardisation. Minimality will again be a useful approach in design, simply because detail of communication is reduced - so it's necessary that support environments can stoop that low. Extensibility appears much involved too, especially the approach leading to object orientation. In the new strategy, any programming language context may be viewed as an object, and programming between languages viewed as message passing between objects. But within any language, neither total object orientation (like Smalltalk), nor total concurrency orientation (like Occam - see Hull, 1987), nor any distribution scheme is really necessary for this purpose. The difficulties in reasoning about language quality imply that language contexts must be held sovereign, and so the implications of outside linguistic requirements really should be kept minimal.

In this approach, however, there is some impact at almost every level of programming practice. The impact on a programming language implementation, and usually on its supporting operating systems, is the need for interconnection and intercommunication both fast yet safe: this is unavoidable. The impact on a programming language is the need for the context opening: this too is unavoidable. While unavoidable, such impact on practice need not be large - indeed minimal disturbance is the intent: no practice in language or in implementation is otherwise of direct concern. The impact on a programming language user is avoidable, at least initially: a new ability, to use or to ignore. The impact on programming itself is the availability of the new ability: to escape any particular language, to span several languages, to transcend - as much as is possible - all programming language. Such programming between language, interprogramming, in allowing arbitrary language escape or recourse, is a general framework for facilitating cooperation and so leading to collaboration.

The framework is built by connecting the communications between context openings in programming languages, and each of these three components needs close examination. Connection must found the immediacy and integrity needed in supporting language implementations in operation together. Communication must found the generality needed to enable crossing between languages. Context Opening must found the general flexibility needed to specifically provide language escape or recourse as desired.

SECTION II. INTERPROGRAMMING: CONNECTION

In supporting interprogramming through all languages, some form of connection is the necessary groundwork for the requisite communication. The connection must allow efficient communication, as did reliance on some common implementation, yet also provide assurance of integrity. Moreover, to minimise impact on any programming language implementation, and so provide generality enough to encompass all, no sweeping requirement is allowable for common structure beyond that strictly necessary. Of course, all these are matters of degree, and while there may be some variance in the precise provisions made in separate implementation environments, this need not be of general concern as long as compatibility can be maintained at least at the low level.

The efficiency requirement must allow the connection to be made as directly as possible to minimise all overhead possible. The integrity requirement must limit the connection to support passive communication only - no programme should itself manipulate data in another language context. The dangers in manipulation are either of expectation of some general manipulation arrangement that encompasses all circumstances, or at least of trust in peace and sanity in what manipulation arrangements there are. Both are unrealistic. The requirement for generality must imply the connection allow only any passive communication but other without structural interference.

Provision For Parallelism

Connection support should ideally provide some protective envelope about the framework of any particular language, a kind of envelope common to all languages in implementation. Such a structure might be very simple and for this purpose only, or perhaps part of some other partitioning scheme of the implementation environment - perhaps indeed used in various ways in language implementation itself. Most independently, a new level of software might be interleaved beneath implementation particular to any programming language yet above implementation common to all programming. While this might be a reasonable but hesitant first step, management of differing programming languages while assuring integrity or even security would prove difficult. For these reasons, and for efficiency, the correct level for an interprogramming bond is the operating system - especially because operating systems and underlying machine architectures are usually themselves working closely together. Operating systems are often large, proprietary, impenetrable or otherwise frightening: all reasons to avoid unnecessarily considering bending software to new purpose. But while some workable scheme may be possible done independently and a level above, any strong requirements for integrity and efficiency suggest this new purpose best requires provision at a lower level. Indeed, the concern for provision of parallel divisions is quite familiar in operating system design.

Several approaches are familiar in considering provision for parallel divisions of concurrency. Some pseudo-parallel computing has long existed to combine processor executive support with application programming, affording separation from application concern with the detail of the operating system. Control

is there determined by explicit demand, one way or the other. More generally applied to application programming, this structure might be seen as a set of coroutines. Simple "multiprogramming" originally arose then as a method of sharing a single processor among several independent programmes, allowing simultaneous access for several applications. Control is there determined usually by some scheduled sharing of processor time. While this offers much separation, it denies much else. But "multitasking" further caters for communication between otherwise separate processes, so creating an accessible structure often seen useful, especially in real-time control or modelling applications. Control may be determined by scheduled time sharing, but can be implicitly or explicitly influenced in application programming. All these approaches may maintain process integrity (through use, for example, of parallel address spaces), but also may not. This is a decision for any particular design, and integrity is sometimes less of concern than efficiency otherwise possible.

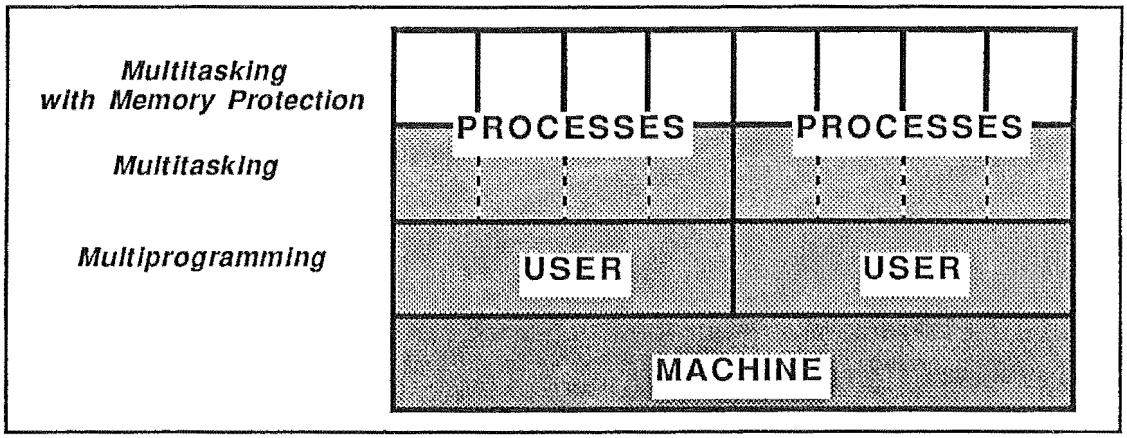

Figure 5.2.1: Levels of parallelism: multitasking is really required for interprogramming, though memory protection is also desirable.

All these approaches imitate parallelism, but the real thing is also available and of interest. Sharing direct access to common resources, multiprocessing might be used in similar ways to any of the pseudo-parallel approaches. And might face similar design decisions: there is much choice still in control and integrity. Less so in distributed processing or even networking, where the implicit allowance for independence makes integrity important anyway, and where control is assumed truly parallel but for explicit communication. Yet there are still a variety of higher level structures possible.

These facilities are about sharing, and the issue that must resolve the design is the degree of sharing required in interprogramming. In fact, the connection requirements could be met through only provision of coroutines with assured integrity. Even the inclusion, in multitasking, of the "programme counter" within that integrity is unnecessary because interprogramming need make no claims between openings in context. There need not be any imposition of any "object orientation". And there is no reason to tackle all particular detail of procedural convention of such structures, their birth, life, and death: it is another linguistically contextual concern and with its own diversity. However, where other inclusive facilities are part of the operating system design already, there is no reason for interprogramming to ignore them as long they

can be limited where languages feel that desirable. The concern of interprogramming is the bridging the gaps between programming languages, first by connection bridging the ones between their implementations. And while first attention must be to those implementation gaps most easily bridged, within a single user environment say, gaps widen from operating system to machine and even computing model. Bridging these immediate gaps should prove practical and indicative, however, and immediate concern will be limited there.

This immediate concern is also restricted to the scope of a single user, but that restriction should also be lifted to span all programming. Programmers might reasonably be seen responsible for the actions and interactions of their programmes, but in implementation of programming the multiplicity of levels mean that programmers are not really alone. And so for actions or interactions in that implementation, the programmer cannot be seen responsible. Every use of another's programme is an act of trust, both in the maintenance of robustness in the face of berserk behaviour, and even in the lack of danger from malice. The former can be guarded against somewhat by extending the protection between users to ordinary programmes, as is done in Unix where all programmes are processes - the same structure that divides users. The latter can never be totally eliminated, as Thompson (1984) points out, without foregoing the community that makes the levels of support software as useful as they are. Many operating systems, however, do not make even the distinction required to prevent very common accidental chaos. This is a reality familiar to many who find their bug-ridden new programme has severely disturbed their entire programming session.

But while not providing exactly the most suitable facilities for provision of integrity, many operating system environments do provide pseudo-parallelism for multitasking and some even provide the passive communications facilities desirable for interprogramming. In understanding what is commonly available, a historical perspective is necessary. A single user might once have meant only a single programme, at least at one time, and multitasking was not of interest within the confines already imposed by multiprogramming. And within multitasking anyway, non passive access relying on common implementation might not only be desirable, for efficiency say, but might even be reasonably dependable too - assuming all stays at the same programming level. But even in the conventional framework of programme interaction, better facilities have proven successful. The Unix facility provides reasonable integrity with the more accessible process level, and then communications facilities between them. And by implementing any particular programme as a process (or a set of processes), then that design can easily do without any special provision for a control language, can provide background computing, and can support simple multitasking. Interprogramming is only a step away.

Until recently, the dominance of multiprogramming operating systems for general purpose computers would have meant that at least some facility was available for establishing programmes (and so possibly in different languages) in parallel. "Personal" computers, however, their low cost and following popularity, have lead somewhat away from that character of general purpose computing. One of the facilities of interest in multiprogramming systems is the possibility of some memory separation strategy. That is structure which most personal computing environments have simply done without totally. However, because such systems are oriented about a single user, this case reduces to that where the user programmes can interfere with each other - common enough even in multiprogramming. The integrity risk of interprogramming would not be significantly different from any programming in such an environment -

where typically even the operating system is subject to interference from wayward user programmes. More significantly, in operating systems for early personal computers there was no support for any concurrency at all. It is possible to cope with such situations with some coroutine arrangement. But more significantly, as user multitasking has become more common in popular larger operating system design, and as the cost of more sophisticated hardware has fallen, so more recent personal computers also encompass the facility. Moreover, the continuing development of personal computing suggests that there will be less and less difference in the programming environments in small-scale and in large-scale computing. And the movement is clearly toward the sophisticated rather than the primitive.

Coordination Of Concurrency

The detail of connection, with some assurance of "concurrency" of sufficient integrity and efficiency, is the precise means of control and coordination within an environment supporting some approach to multitasking. There are many methods of detailed control, Andrews and Schneider (1983) present a wide survey and discussion of the related approaches in concurrent programming. Some are not really directly applicable for interprogramming, involving direct manipulation coordinated by semaphores (classicly after Dijkstra, 1968ac), access capabilities (discussed earlier by McGraw and Andrews, 1979), and other strategies reliant on a broad uniform environment spanning the parallel elements. Andrews and Schneider see two classes of method stemming from earlier methods: "message orientation" such as involved with "ports" (after Balzer, 1971), and "procedure orientation" such as involved with "monitors" (after Hoare, 1974). They then suggest that the newer "operation orientation", such as involved with the Ada "rendezvous", as a joining of the two classes. In fact a number of programming languages have been designed with particular attention to concurrency and control, especially for distributed programming, and are of interest in interprogramming design. Andrews (1982a) has also surveyed these embodiments, and presents (1982b) an operation oriented language, SR, generalising from the procedure design of Brinch Hansen's DP (1978) and the Ada "rendezvous", in provision of a symmetrically structured "remote procedure call" facility. The Ada "Rationale" (Ichbiah et al., 1979) has an interesting review of the situation at that time and suggests that procedural structure was thought useful, but that strict exclusivity was too strong for some real time applications of interest.

The viewpoint in such design is not quite the same as that needed for interprogramming: Andrews and Ichbiah were concerned in languages with distributed programming, whereas interprogramming is concerned with programming in distributed languages. However, in designing particular facilities for particular languages, the experience is certainly relevant. For example, the two general approaches of interest are those that involve "messages", "sending" and "receiving"; and those that involve "remote procedures", "remote call" and "remote return". The first structure is a one way and two pointed send-receive, and the second structure is a two way and three pointed request-receive-reply. In interprogramming, this latter pattern might seem most appropriate, introducing to languages with subroutines or coroutines the possibility of generalisation to remote subroutine or remote coroutine: so perhaps "subprogramme" or "coprogramme". The greater generality of "coprogramming" in allowing call spanning data management might be important in

maximising flexibility, and so interprogramming really should encompass both. Also, it seems to offer a potentially appealing structure, allowing movement freely back and forth between differing programming language environments in a single thread. And finally, because it is a possible connection, it should be provided for sake of generality.

However, while interprogramming might seem similar to schemata for inter-process communication, it must be a higher level of abstract concern. In concurrency, for example, it should not directly be of concern whether separate process structures are used, or separate coroutines. In coordination, it should not directly be of concern whether call-return or send-receive, or any other similar structure, is employed. While practical proposals may point out how a particular structure may be a practical building block, the actual practice need not be a concern. Both approaches may be modelled within one another, neither demonstrably more primitive; other approaches might too be equivalent. This was suggested by Belpaire (1975), and more recently Staunstrup (1982) argues both separately have advantages. Scott (1983) agrees, and uses this as an argument for leaving synchronisation out of language design all together. Interprogramming does not have any mandate for such a strong claim, though within language design is another matter. It's tempting to see message passing as more fundamental because of the axiomatic way that graph structures are often seen as edges joining nodes - but there is no knowledge that more structure does not underly that model.

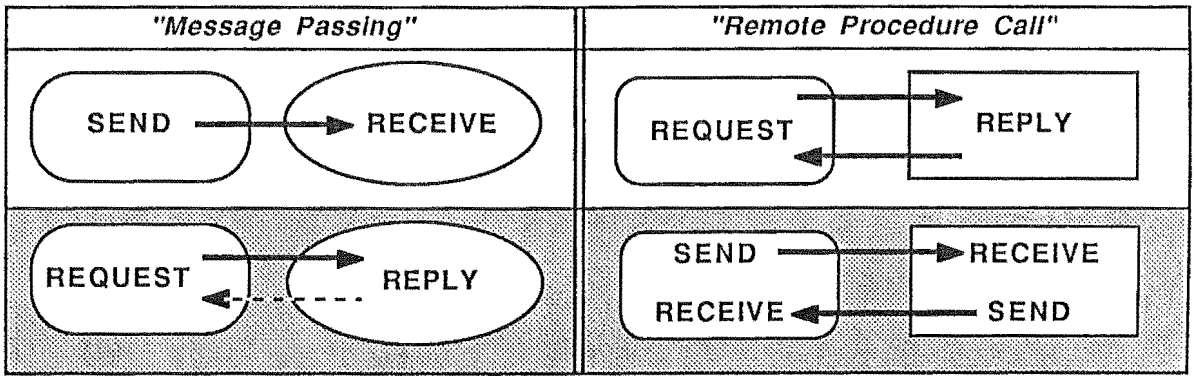

Figure 5.2.2: Different coordination models: at a low level, remote procedure call can be modelled by message passing primitives, and vice-versa. Many such facilities can be modelled by each other, and with no absolutely clear standard, an interprogramming strategy should esure low-level compatiblity with all.

In implementing the coordination for communication, some explicit structure of message passing or equivalent facility is clearly most directly useful. However, it may be possible that shared memory and shared control may be adequate as a base on which to build such a facility. Operating system input/output - a file system - might be used too, even if interprogramming communication need use files as commonly accessible storage media. This might seem to ensure unacceptably poor performance in operation, but in many cases operating system input/output might be sufficiently well buffered and controlled so as to avoid this; in other cases, prototyping for example, the slowness might not be intolerably bothersome. In controlling interprogramming connection, any message passing facility will itself be a basis. In systems organised differently, however, some other device - semaphore, lock, signal, may well meet the

requirement. Again, operating systems input/output may well be of use again, and the coordination requirements often already provided in their operation might be used piggyback - re-opening a file for exclusive writing, for example, would assure integrity.

The coordination detail involves the particular matching of context openings in differing programming languages. To so match, contexts and openings must be distinguishable in some way. Again, strategies employed in analogous computing situations vary. Many are "label" based. In some cases, any two communicating elements must know each others label. This is assurance, when important, as in the CSP (Communicating Sequential Processes) approach of Hoare (1978), of mutual comprehension and tractability that may be accorded. However, such assurance is not strictly required in practical computing, and would unnecessarily limit the application of interprogramming - though only linguistically, analogously to the way "if" and "while" are more limited than "goto". Most approaches require only that the initiator know the label of the respondent, and as this is the minimally restrictive strategy it is indicated in this case. In some approaches, labels are bestowed uniquely at element creation, thus matching might be done by or at least through some creation environment, whatever that might be. In other approaches, any labels are self imposed. In this way labels and label use can be easily integrated into any available and appropriate programming language context.

Fraser (1971), in a proposal for integrating different programmes within an operating system, shows how one hierarchical arrangement could span both programming language and operating system name spaces; languages that encompass operating systems, as is suggested with Smalltalk (Ingalls, 1981), also use an integrated approach. The interprogramming intention, however, is to allow programming language design as much freedom as possible, so more flexibility is called for. Openings need not be identified by "label" at all, with the natural language flexibility that implies. It would be sufficient for openings to be "numbered", were that suitable in context, or indeed distinguishable in any way. This might be seen as a generalisation of label definition and reference within compilation units - typically so important when relying on common implementation for cooperation. In this more general case, through such matching only is access available, and in such matching any distinction is available. Such a technique can thus involve a particular language independent of any particular support environment, though interpretation and actual matching must of course be an implementation level function. This matching capability might span levels of programming scope as appropriate and reasonable within the implementation environment. In this way, such designation is similar to that frequently made involving programming language input/output. Any programming language might provide labelling or other designation for matching, but the consequent matching itself need not be bound.

The practicality of coordination for interprogramming seems reasonable. Some form of multitasking is the main ability that needs be provided, preferably with memory protection between parallel contexts. The former at least, and more and more the latter too, are becoming commonplace in operating system design. Various detail coordination geometries would be sufficient, certainly a precise message passing arrangement would be sufficient. Some flexibility can be affording in labelling strategies as they could be externalised from any particular language context and converted and matched from the outside. Clearly any standardisation in these structures will assist any language and supporting implementation for interprogramming, but initially the flexibility is more important. There is neither enough practice nor enough

consensus to justify influencing language design very much. In working towards language collaboration, it is the language design that should be leading the evolution. Pragmatically, however, popular design in support systems and attendant efficiency and ease of distribution will of course have great effect, so some interplay between languages and support system design must be expected.

SECTION II. INTERPROGRAMMING: COMMUNICATION

Simply connecting programmes to allow communication requires some preparation, but the mutual comprehension required in communication itself requires even more. So few concepts span all programming languages, indeed, it might seem impossible. At a lower level, the problems are similar to those encountered in establishing programme portability to a new supporting environment (see, for example, Inglis and King, 1977, for a discussion of such data influenced portability). As made clear in detail by Kent (1978), data structure at a higher level is in the eye of the beholder - integers, arrays, sets and all are human inventions. However, there is one fundamental and spanning foundation, primitive concept though it may seem, and on this foundation communication may always be established. In application and implementation, it is the bedrock of computing, of digital computing, that computing is digital: representation is discrete and exact. This is the assurance that communication at least at a low level is in fact always at least possible, and that can be built upon.

Standardisation

The problem of standardisation at a low but digital level is of concern elsewhere, the problem being the interplay between flexibility and efficiency. In virtually all digital computing, the most basic digital encoding used is the same: the bit string. This is the finest granularity precision bound on flexibility, so if sufficient efficiency can be assured, this seems the desirable direction. In digital network communications, for example, at the first digital level of structure above the physical, the "data link" layer, standards have moved away from association with "characters" and "character codes" to a bit orientation in protocol design (for history, see Tanenbaum, 1981). In this way data transparency is possible between differing structure standards at least at one digital layer, so enabling the possibility for communication at some higher digital layer. However, in structuring not in application but in implementation, data transparency is still possible. Whereas earlier character protocols, like that of Arpanet, relied on recognising certain particular "characters" and provided transparency through "character stuffing", the newer protocols, like the ISO HDLC, rely on recognising certain bit sequences, and so provide transparency through "bit stuffing". This enables any required flexibility in connection and coordination. In operating systems general purpose file systems, too, it has similarly been suggested that the bound of bit-orientation might have advantages (see Lions, 1983). The difficulty of flexibility versus efficiency will persist, of course, but the generality of reliance on the bit bound might make it increasingly less relevant. But while the conceptual access to the bit level might be preserved, as in the ISO standards, there might be acceptance that the realisation of the precision bound is more coarse. The popular 8 bit unit is accepted as that granularity, hence the ISO recognition of dealing with bits in "octets" regardless of the physical implementation, so making the most common implementation the most overhead efficient while retaining generality.

Of course, few programming languages are explicitly much concerned with such low level encoding as the bit. Even so-called typeless languages orient about some more complex though uniform structure. In which case, the level of atomic indivisibility is what must be determined, and it is the bit that is

the lower bound. Most languages involve several different more complex structures, often upon several levels, and many more recent languages cater for definition of new types in application. As such structures are all eventually specified in bits, the question of data standardisation arises first there. Standardisation, however, even at the data level echos the problems of programming language standardisation discussed earlier.

The most significant data standardisation effort - earliest, largest, most successful - is that concerned with the "character". Communication is possible with only ad-hoc agreements, but the volume of character oriented communication between computing components is so great and continuing that efficiency is a great and continuing concern. Standardisation of representation for characters has been quite usefully successful. This case might seem superficially be such a straightforward case for standardisation with success that it deserves some examination. The standardisation is not total even now, the question cannot be regarded as closed, and it is not even clear character standardisation is firm enough ground in the context of interprogramming.

Firstly, there are questions about the particular standardisation achieved. In fact there are two widely supported standards, Ascii and Ebcdic. Both have a clear basis for popularity and continuity: Ascii was designed and is followed by a large number of different computing suppliers; Ebcdic was designed and is followed by the single largest and often dominating supplier, IBM. Neither's popularity seems clearly on wax or wane; both have some advantages, some disadvantages, dependent on context. Moreover, standardisation or even compatibility is not always in commercial interest. And even within each "standard", there is sometimes surprising divergence. In some cases, variance is necessary to encompass character usage in local orthography. In other cases, variance is a technical expedience in limited computing circumstances. In yet other cases, variance is simply a measure to attempt compatibility with rare or historical non-standard representation. MacKenzie's 1980 book presents a historically oriented review, and details what a huge effort data standardisation is at this one level alone.

Secondly, there is the larger question of character standards at all. The importance of the "character" is that it is the link from computing to general notation, writing, and computing character standards have largely followed precedent. But precedent, of course, is not itself standard. Accordingly, character standards follow the precedent of commercial application: European writing, particularly English, more particularly American English, and that of the business community. And so computing in any other context must adapt to the standard, or venture into the non-standard. Adaptation has often been the case, but should computing continue to diversify, so will ventures into the non-standard, and so will standards change. Even in applications seemingly well served by existing standards, opportunities presented by increasing computer capacity and decreasing cost may seem inviting. At one time the lower case alphabet might have seemed a luxury. At what time will any single and fixed width font seem an intolerable restriction of programming language design? With the equipment now available, the time will likely be soon.

A more recent attempt at establishment of low level type standardisation is that addressing floating point numbers. The work was coordinated by IEEE (1981), is now an Ansi standard (1987), and considers formal mathematical implications and numerical practice, as well as portability. The effort is recent and new

in application, though though the design is supported by a reasonably broad selection of the computer manufacturing industry, so some success seems indicated. In numerical work there is perhaps a more widely accepted criterion for representation, and perhaps less to be made of divergence than convergence. But the new standard is novel in several ways, and existing facilities, and use made of them in programming language design, seem comparatively primitive, so full adoption in software will not be trivial (see Fateman, 1982). Anyway, investment in existing numerical practice is great, some of it embodied in hardware, and may slow acceptance of any new standard. One impetus for the cooperation has been the wish to incorporate floating point capabilities into anticipated new microprocessor architecture. Old architectures do persist however, especially in larger computers, and the importance of compatibility with older floating point practice is still very important. And the commercial aspect of diversity must again be remembered. Progression to complete standardisation cannot be assumed.

Whatever detail standard is used, of course, may be rendered not immediately compatible through use of encryption, compression, or even structuring techniques. It may be assumed that antidotes exist, but standardisation becomes more clearly difficult to achieve in general.

But while not sufficiently dominant to be standard, many practices in data representation are very common. In the representation of integers, for example, fixed length bit sequences are most common, and two's complement the next level of encoding. Conversion with the other standards at this level, one's complement and sign-magnitude, are simple enough. At representationally higher levels, other structures also have many practices that are widespread - the representation of arrays by sequencing in lexicographic (or reverse) order, for instance, the sequential representation of vectors, and strings. Though so commonly forming basic building blocks, such strategies are not unique - compare the bit or byte order used in the IBM-360 et al. with the PDP-11 et al. However, such common practices are important standards in the close community of design, especially that of a processor or operating system family.

Despite the problems in deciding on standards and on establishing them in usefully wide practice, standards remain of great importance. Probably the most useful attitude to take is one strongly supporting standards where experience has evolved to approach consensus, and to leave alone areas with wide gulfs in practice. Much work continues all the time. At the low levels of data in programming languages, Darondeau, Le Guernic and Raynal (1981) suggest standards that could apply across languages. The problem there is mostly from practice working closely with individual machine architectures. At a higher level, ISO supported work on an entire "Document Architecture" (Horak, 1985) also encompasses standards for some data types common in programming languages. This work addresses larger units of data, and because creation of document data in that format would be likely be handled specially, even the low level elements would not present a particular problem. However, this also means that the scale would really be wrong for the immediacy important in interprogramming. The lower level components forming the "character content architecture" would be still be germane to low level data types, but it's not clear that the standard will have a sufficient impact at that level generally. It is not even clear whether the time is right for a cooperatively developed standard for whole documents anyway, because of the advent of image description languages (most notably the de-facto standard PostScript - Adobe, 1985), and their considerable impact on document representation.

All disagreement, all divergence, all variance, make dependence on standardisation a limiting decision. Compatibility with any standard is a clear advantage, but reliance on any standard is a clear disadvantage. The importance of efficiency in any programming suggests some recourse to standards: the importance of flexibility in interprogramming suggests something more as well.

Translation

In supplying the support for communication in interprogramming, some attempt should be made to take advantage of such similarities while standing on guard for differences which still sometimes occur. Adjustment and conversion of encodings might be handled specifically by one or other communicating programmes or programming language environments. While this might indeed be straightforward for some programming languages it is certainly not appropriate to all. Moreover, it is rather an imposition on the programmer at this time and at this level. Moving up in abstraction, use might be made of the common practices that do exist in computing environments for encoding or decoding - numbers to characters etc., and many character based utilities. These practices have a long precedent in application to files written by a programme in one language and read by another. A common standard is indeed to make everything based on readable characters, simply because it is the one common representation that most languages have a reason to be able to deal with - for human use. In this way, when communicating languages or programmes share common ground it is to their mutual advantage.

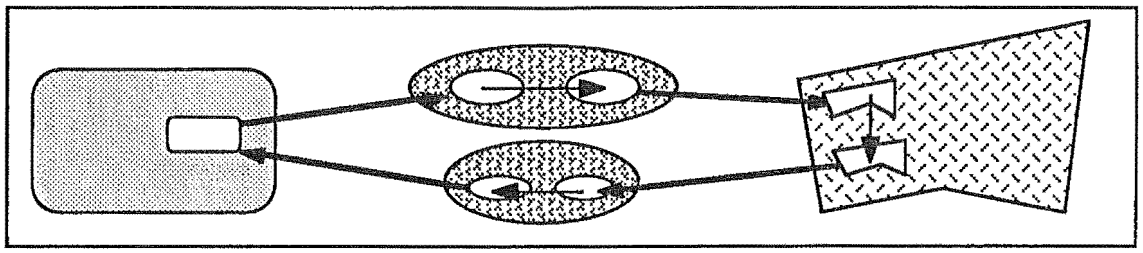

Figure 5.3.1: Programmes in different languages with incompatible data structures could be linked with programmes in a third language - perhaps designed for that purpose.

More subtly, communication from source to target might proceed via an intermediate envelope in an intermediate language. With appropriate and powerful enough "interprogramme" and "interlanguage", such a scheme would be quite general. While languages powerful and explicit enough for such programming do exist in most programming environments, the context of discourse might still seem inappropriate, too wide or too narrow, and a new language might seem worthwhile. Moreover, should such a facility be offered in conjunction with the support mechanism for connection, the operating system for instance, communication of such generality and flexibility might not require great sacrifice in efficiency.

In connection, a support strategy similar to the "remote routine" approach seemed appropriate. In that approach, the analogy with more familiar practice with subroutines and coroutines is sometimes furthered by the typing of parameters. In communication, this is a nuance below or above the capability that can be assumed. Type is a guarantee of malleability, and across the boundaries between programming

languages, no guarantees can be demanded or honoured. While this does mean that there is no assurance that programmes will not encounter difficulty should data formats differ, it is no different to existing practice with regard to programmes and, say, files. Of course it is still possible for programme environments to vet incoming data, and it is still possible to make outgoing data easy to vet. In Unix, for example, files with unusual formats include a "magic number" for rough but quick verification that format intention and expectation match (see, for example, the entry for "a.out" in section 5 of Leffler, Joy, and McKusick, 1983). In a similar way, implementations of Modula-2 verify proper matching of separately compiled parts by means of a "key", an arbitrary number that, like a ordinary key, doesn't completely deny malicious or accidental access, but makes it more difficult and less likely (see McCormack and Gleaves, 1983). Of course too, in designing software it is as well to keep formats compatible where possible, and the more so obviously the easier is interprogramming communication.

Of course programming languages might well and often do limit objects in input/output, and so might do the same in interprogramming context openings. Output of a "procedure", for example is often not provided for. Even what is commonly provided might itself be seen in different light: output of "variables" might, as in the PL/I "Put Data", include both "name" and "value" constituent. Moreover, input/output aside, what constitutes a programming language "object" is itself in question: what can be communicated? Constants? Variables? Expressions? Procedures? Statements? Only language definition itself can answer. The answer affects any programming, certainly interprogramming, and drives much detail of implementation. But it isn't a new problem. And because in practical interprogramming application domains must clearly overlap, that requirement on the discourse domain need not seem too strong.

With appropriate support from coordinating structures, and through openings in language context, interprogramming communications should always be possible simply because of the basic digital structure involved. Standards for basic data types would make actual exchange of data from programme to programme much less of a nuisance, however. But standards being difficult to either justify or establish, some data translation facilities are also called for. These could be handled internally to some languages, but not for all, and some data translation language would be useful in these circumstances.

SECTION IV. INTERPROGRAMMING: CONTEXT OPENING

The operational foundation of interprogramming, connecting and communicating, must be beyond any language context. To use this foundation, however, to interprogramme, must involve opening language context itself. Both lace and eyelet are essential. In specifying any opening in the fabric of a programming language, however, great care is indicated: it is this level, by definition, that is the context of actual day-to-day programming. Great care is indicated in integration of language opening with language, by design and by application, in style and in community - as much care as lavished on any other aspect of language design. In total isolation, a language is a world apart. A lost world, unreachable, unknowable, and almost unusable. For in use, programming language is almost always at least partially open.

At an abstract level, there is an opening inherent in programming: an application/implementation opening. For computationally compatible languages, a programme in one language may represent another programme in another - with computational equivalence. This "opening" is clearly of fundamental use: almost all practical computing depends upon it. On a higher level, several already considered strategies for programming language cooperation depend on it: Translation, Inclusion, Extension, Common Implementation all rely on representational equivalence. In interprogramming, however, it is difficult to see direct application of the opening without the difficulties limiting those same strategies.

At an immediate level, there is an opening implicit in programming generality: an entry/exit opening. In any programme there must be recognition of transformation, always a recognition of transformation application, always a recognition of transformation limit or continuity. In operational terms: every programme has a beginning; most have an end. In functional terms: every function has a domain and a range. In recognition of such transformation, there must be consideration of the surrounding context. In operational terms: something will precede every programme; something will follow most. So in connecting programmes, the limits - the beginnings and endings - are openings suitable for communications. This opening is frequently used in practical computing, though almost always in a quite limited way. This usage is the basis of some strategies of cooperation discussed as Programme Coordination, the limitations then seeming bothersome. In interprogramming, the limits would certainly prohibit the flexibility possible were this the only opening.

On a practical programming level there is an opening implicit in programming practicality: an input/output opening. In practical programming, computing is for effect, and without input or output there can be little effect. However, there are many views of input and output. Input might simply be regarded as integral with a programme, output might be seen as the effect. Programme control over input and output varies widely, and is often very limited in more specific application programming languages. Even in more general languages with flexible and detailed input/output capability, any particular implementation and instance might exert influence through buffering or data mapping. This opening is of course a common part of day-to-day programming. In programming language cooperation by Programme Coordination, some strategies are based on the opening, though commonly face severe restrictions. In an earlier approach to interprogramming (Biddle, 1983), input/output was seen as a sole foundation for the necessary

communication without dependence on any particular language or (much) on particular language implementation. The capability has surprising potential, as Unix practice shows in pipeline direction of the "standard" output of one programme to the "standard" input of another in parallel. But there are limitations in flexibility and some conflict in orientation of design and implementation of detail. However, should language input/output offer the necessary flexible precision, it alone might prove a usefully general opening. Suggestions for general abstractions of input/output in programming language design have been made by van der Boos, Plasmijer and Hartel (1983), and by Mulders (also 1983), and would be a reasonable, though not an exclusively reasonable, approach. The general and orthogonal methods they proposal are interesting, but still linguistic in character. Pyle (1979) discusses the importance of programming languages having more complete control over input/output, so to more easily encompass differing devices and idiosyncrasies. This is nearer the capability desirable in interprogramming, where the "devices" would be other programmes. Pyle goes on, however, to advocate input/output modules be isolated for different cases, and suggests Modula and Ada illustrate the right direction. In interprogramming, there is no reason particular languages shouldn't offer input/output specifically tailored to some particular application area, or even just in idiosyncratic exploration. As long as they can encompass simple passive data communications, with simple control over the buffering granularity of communication, they would be sufficient to communicate with other language programmes.

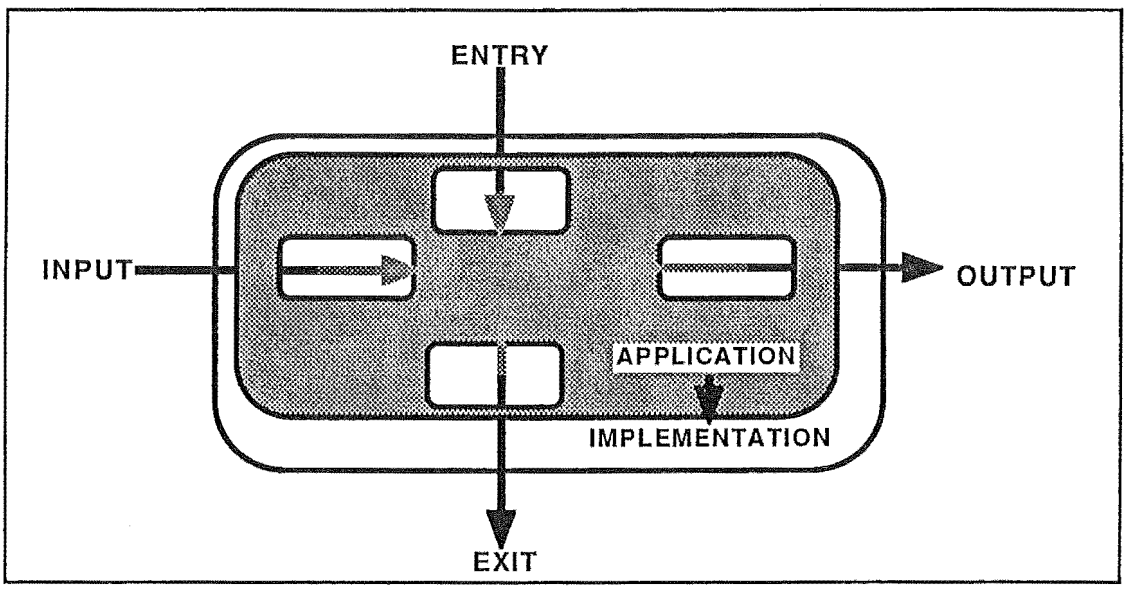

Figure 5.4.1: Most common openings in programming language contexts - "Opening" is used as a generic term for language structures that involve escape beyong the language. Input/Ouput is the most usually the most explicit in language design, entry/exit is often tailored to a particular environment supporting the language, and application/implementation simply acknowlegement of the implicit openings depended on when relying on a particular basis of implementation.

While some of these openings discussed are often implicit in programming language design as the facilities transcend the limits of language itself, they might also be made explicit in acknowledging such limits. For instance, languages may have facilities incorporated to accept information as they begin, and

return information when they end. Such acknowledgement might greatly facilitate any interprogramming, and certainly understanding of it, but it is a language decision.

In addition to the most common openings, several others are in wide use. As discussed earlier, probably the most common present strategy for cooperating between languages is the Common Implementation approach making use of separate compilation. In this case, the call-like and procedure-like structure might be regarded as implicit language opening. While some languages make pretence that separate compilation is only to be done within the language context, other languages acknowledge the opening in the fabric and even make provisions for other language aspects, both for specific other languages, and more generally - as discussed earlier when looking at cooperative strategies relying on common implementation. Where the practice is not encouraged, there are often good reasons involving memory management or procedure format. Where such use is expected, programme language implementors might consider the ramifications of unusual implementations, but clearly the final responsibility must rest with the user. There remain all the problems of parameter formats and the detailed protocol: the opening is useful but often not very general or flexible. Tsin (1982) proposes a kind of general procedure call as a solution to access problems from Pascal, using the common "external" directive for procedure declaration. He suggests a pragmatic "Z" option that makes the indicated procedure known as not (necessarily) Pascal. Integrity remains a large problem, as does the difficulty in communicating between more complex programme units.

Sometimes viewed also as similar to "procedure call" is the structure in many systems programming languages used for communications with operating system executive support: the "system call". There are many mechanisms fulfilling this function, though it might be viewed as an general opening in any programme, and what happens beyond viewed as quite unknowable. Unlike with "procedure call", there is here some assurance of integrity and security in the communication, though by far most typically only one-way. And aside from assembler languages, there are few programming languages that make this opening explicit rather than incorporating it implicitly with procedure call. Offering a less specialised opportunity, some computing packages offer "hooks" allowing a user some limited access to data in mid processing. In sorting and merging facilities, for example, one might have access to data under inspection as required.

Perhaps the most blatant opening ever included in language design is that in some systems programming languages allowing direct insertion of arbitrary material into the target translation of programmes. Such an opening allows anything the target context does, usually a great deal - including much potential for chaos. While obviously open to abuse, this facility does reduce the need for or reliance on assemblers, and does allow great efficiency at critical points. Remembering separate compilation is also likely available, and that there is likely little protection between procedures, the abuse possible is in fact easily achievable in other ways anyway. However, in interprogramming, the flexibility afforded is of course not worth considering the loss of integrity or security: all the difficulties in "procedure call" opening, only worse. In Zed (Cheriton et al., 1979), the keyword introducing this opening is "twit": the window into trouble.

Perhaps the most subtle opening is that involving "pseudo-variables". Such might syntactically appear as a variable, but in evaluation or in assignment allows communication outside the language context. In Snobol, for example, variables established as "input" and "output" may appear in any situation that any

In Snobol, for example, variables established as "input" and "output" may appear in any situation that any other variable may. But hardly with the same results, as any evaluation of "input" yields the next line of an input sequence, and any assignment to "output" produces the next line of an output sequence. In ways similarly implicit, other language contexts might be opened by appropriately blessing otherwise unused variables. Formally this is dubious, of course, but it works well Snobol practice.

Few of the widely available language openings provide all the interprogramming flexibility that might be wished. Following discussion about language connection and communication, however, it should be clear that more general message passing or remote procedure techniques might well measure up. But so explicit techniques are neither widely involved in many programming languages, nor widely standardised themselves. In considering design specification for some new technique, several components of the approach might be considered. For instance, as discussed, there is a choice between the message communication primitives and the remote procedure communication structure. Whereas "send-receive" is a two-cornered set, remote procedure might be a strict two-corner "request-reply" set, or a three-corner "request-receive-reply" set. And if a remote procedure call model is chosen, languages must then address the definition of any necessary nesting rules. Then there is the question of whether communication need be via any intermediate mechanism, such as mailboxes, or whether communication might be direct. And even then the questions of data and of data bundling still need to be addressed.

However, even less specific commitment is required than in connection or communication, because great flexibility in choice is implicit in the approach. Any particular programming language must encompass some particular design for opening context to enable interprogramming; but no particular design need be imposed in detail. In this way, whatever form a particular language opening may appear is a concern for designers and users of that particular language. There is no reason why interprogramming need be or should be involved in such design, save the minimal specifications it must demand. As a beginning, once assured of security and integrity in implementation of connection and communication, specification for language context opening could simply be flexible by being minimal.

SECTION V. CONCLUSION

As interprogramming offers arbitrary escape from and recourse to programming languages subject to their support in opening, it is a maximal solution to general programming language cooperation. As interprogramming unites extra-language access in a general independence, it offers maximal flexibility. The framework involving coordination, communication, and context opening, does seem workable, and it ties together several directions in software design.

The coordination required should be reasonably easy to provide on any operating system that supports user multitasking. Memory protection of such parallel programme environments would be useful, but is not essential. Several detail coordination mechanisms would be sufficient, and simple message passing seems the most straightforward to build upon. And no one particular labeling strategy is necessary, assuming that some form of label matching could be done externally to particular programmes.

The communication between different language programmes would clearly benefit from increased adherence to standards in data representation and implementation. However, the digital nature of the low level of implementation involved means that communication is always possible in principle. Moreover, data representation and structure cannot, at the higher linguistic level, be standardised and not impact language design without justification - even simple data structures have proven difficult to standardise completely. Some tools for data translation should help to overcome these problems, both within languages and as new "languages" themselves.

The openings in the context of programming language through which communications pass are very much dependent on the individual languages themselves. However, the entry/exit and input/output openings that many languages already have may often prove sufficient or almost so. Even simple new procedures would be an effective, though primitive, way of opening language context. The design questions seem likely to centre on the relative merits of well integrated openings, and on openings that are easy to introduce without changing the language. New procedures might be easy to add, for instance, but there would be difficulty in static type checking of their arguments.

So, overall, the approach seems worthwhile and workable. However, as consideration of convergence possibilities might indicate, interprogramming cannot be not a magical immediate solution to all facets of the diversity problem. At a linguistic level, interprogramming cannot directly address structure internal to a language context, not at all aware of any such context. So, for instance, a Pascal "while" loop cannot directly involve a C "break" capability. Interprogramming must pivot on language ability to isolate component structure, and can only succeed as far as this commitment is made. It is an important strength of the approach that both coordination and communication are passive, because it is that factor that allows programmes and languages freedom in design. Whatever design is chosen, as long as a commitment to context opening is made, it can be interfaced to other programmes and languages outside the context. However, such external interfacing is an extra programming task and implementation concern, so interprogramming will clearly be eased where standards between languages for coordination and communication are established. But with the capability of external interfacing possible where needed, design will be able to evolve such standardisation while and through actual practice. Because agreement necessary

for standardisation is difficult to reach, this evolutionary capability is important. It is also an important difference between interprogramming concern with standardisation, and the advocacy for standardisation in subroutine linkage conventions so to enable "mixed language programming" by common implementation. Moreover, because the standardisation focus, the "opening", is not involved with detail design or implementation efficiency within the language itself, the chances of standardisation success appear greater still.

At an implementation level, interprogramming cannot be as efficient as programming within a single language. Maintenance of connection integrity and passive communication is not without cost: it is always faster to do things directly if possible. Memory context switching for integrity is often of critical importance in efficient machine design anyway, though, so performance may be sufficient for all but the must critical applications. Passive data copying is more problematic, but such is no worse than required for input/output. And use of an inappropriate language may not itself be without its own costs; some convenience is worth some cost. While not directly applicable to most computer architecture currently in use, a more sophisticated approach to memory protection such as "capability" oriented architecture (see Wilkes, 1982). might resolve this dilemma. With a sufficiently flexible application, it might, for example, become possible to pass data from one context to another without the possibility of breach of security or integrity, but without the need to copy.

At an abstract level, interprogramming cannot itself address any philosophy about language closure, open to any context on offer. So, for instance, it gives no grounds to impugn any language or language implementation determined to remain in isolation. And insistence by the implementation basis itself is irresistible. As discussed, total isolation is difficult, but isolation sufficient to create great practical inconvenience is much less so. While interprogramming is an attempt to span language contexts, that too is a context: if such eclectic programming is not acceptable, little more can be said. But if one accepts that language convergence is not about to happen, it seems only sensible to specifically prepare for diversity. The diction used in discussing language diversity sometimes seems biased against it: "proliferation" (Ichbiah, 1984, but many others too) resulting in a "hodgepodge" (Heering and Klint, 1985), for example. But with some preparation there is no reason why a hodgepodge need be displeasing, and proliferation not be seen as a happy result of happy exploration. In their look "Towards Monolingual Programming", Heering and Klint deplore that a programmer need be a polyglot. But if such is being a "polyglot", then the programmer is hardly alone, for many human activities span several such "languages". Sometimes by accident, and sometimes by design, and sometimes distinction between reasons and between languages is difficult. Programming between programming languages need not seem unusual nor unappealing. Clearly the nature of any particular language involved would not be without effect, but again this is nothing new nor necessarily bad.

The possibility of general interprogramming seems sufficiently convincing to consider the practicality: how to proceed. Two levels are involved, the programming language support - the operating system, and the programming language itself. Minimal impact must be attempted in establishing interprogramming, it is as much an absence of philosophy as a philosophy at all. To so limit the impact, it seems best to only add, to add only a little, and not to otherwise alter. In further language evolution, it might

then seem suitable to integrate interprogramming into overall software design. With even the introductory steps taken in existing software, and the possibility of recognition of the approach in new programming language design, some resolution to the problems of programming language diversity should be realised. Later, should an interprogramming approach influence not just software but programming itself, it seems possible quite new programming contexts may arise. Interprogramming is a general attitude more than any specific feature, and the design of tolerance to diversity is itself an invitation to evolution.

The most operational effect of interprogramming is on programming, and that effect is to allow in programme design the choice and application of languages together. Consequently, the effect on programming languages is to free design from the need to offer everything that might be necessary in an application programme. So languages with specific orientations would need no longer address concerns outside their interest yet demanded in practical computing. Some other language, via interprogramming, could meet those needs. In such an environment of cooperation, of collaboration, Cobol might not need floating point arithmetic, Fortran might not need so much data formatting, Algol might not need string manipulation. And any language - famous or obscure - would have access to whatever power, feature, nuance, context, is on offer in the pantheon.

The first step is to look closely at how existing programming languages and operating systems could encompass interprogramming.

CHAPTER VI
COLLABORATION: INTERPROGRAMMING INSIDE PROGRAMMING LANGUAGES

Collaborating between programming languages must require some capability within the programming languages themselves. It does not seem difficult to design extensions within which to house the minimal capabilities requisite. But making such renovations in structure harmonious and in character agreeable to a particular programming language will require subtle design. In application programming, the language itself - as well as any implementation - is of great practical importance. And as with language design, so with language extension: there are many factors to consider, many contexts to consider them within. Again, diversity looms. This language extension, however, unlike previous language design, is itself some proof against just such diversity. And while this is clearly not a reason to invite diversity without bound or care, it should prove reason to trust the limiting to such community as there is in language design, implementation, and use.

In so trusting extension to appropriate language communities, proper emphasis is also given to the important separation in design of interprogramming and of any particular programming language. However, some practical exploration and illustration is worthwhile as examination of interprogramming. On these grounds, the situations of some actual programming languages is considered here. And some variety is called for. In paradigm, in linguistic structure, and in canonical implementation. The historical and structural implications of the "general purpose" paradigm, "procedural" and "compiled", are considered good reason for close examination, so Fortran, Pascal and C are all discussed. More specific in orientation and common implementation, Icon then widens the scope, followed by the less sequential orientation shown in Prolog. Obviously several other languages would be interesting to examine, for both historical and linguistic reasons, and this list has also been influenced by personal experience and local provision.

In considering interprogramming through these contexts, any existing openings will be examined, any necessary expedients for further opening proposed, and integration of old and new considered. Discussed in all cases are the questions involving connection models (send-receive, request-reply) and labelings, and communication data translation and granularity. Some thought is then given to less obvious examples of programming language, and then on that taxonomy itself. All proposals for particular changes to particular languages are intended only as examples in the context of discussion showing how opening might proceed. Partisan feelings may show, and indeed it is difficult to see how they might be avoided. In the following designs for extension, for example, there is possibly too much attention given to structural symmetries. Some proposals are admittedly unlikely to win favour in the language communities they concern; some indeed would be inefficient or lead to implementation complications. However, exploration is the objective, rather than any immediate development or construction.

SECTION I. FORTRAN

In any attempt to span diversity, Fortran seems a good place to begin: were Fortran not included, many many programmes would be excluded. Moreover, because of its early popularity and influence, Fortran illustrates several situations of interest throughout much later language design. In this discussion, concern is with language in most popular practice over time, similar to the 1966 Ansi description. The language of the 1977 description is now itself popular, but the earlier is studied here because of its longstanding dominance in the practical programming language world.

Existing Openings

In Fortran the most common language openings, entry/exit and input/output, are explicitly defined as part of the language. The entry/exit opening is rather spare. The entry of a Fortran programme is not only without parameter, but is also without any distinguishing label, and indeed without explicit form: it is merely a lexical beginning. The exit is only little more wieldy. There is provision for a parameter, though only one and a very limited numeric one, and with little assurance of what might befall it. It does have explicit form, the STOP statement, and flexibility in its placement. If the entry/exit opening were to be useful in interprogramming, much assistance would be required from any other openings available.

Input/output is much more flexible. The facility is explicitly used via the READ and WRITE statements possibly in conjunction with the FORMAT statement or format array. Data may be interpreted or represented with a variety of appropriate mappings easily specified and according to closely controlled layout. Distinction of openings from beyond the language environment is provided for by associating any particular input or output with a particular "unit" number. However, unit numbering is less expressive than might be desired. Moreover, the control of layout provided for is not quite as close as might be wished, the variety of mappings dwelling on fixed format character numeric representations. In control, there is little reassurance of the realisation of input/output with granularity less than the duration of the programme. In mapping, there is a limit of one "character" resolution in any input/output using format, and no control whatever without format. None of this is surprising, considering the intended application. In interprogramming, without further support, this input/output opening is just sufficient, though a bit clumsy to handle and none too sharp.

Fortran also has implicit openings, SUBROUTINE and FUNCTION. By declaring either EXTERNAL, their definition may be hidden, may be outside Fortran, though of course compatibility is assumed. Via CALL to an imagined SUBROUTINE, or reference to an imagined FUNCTION, parameters could enable both coordination and communication beyond the Fortran environment. Control over CALL is precise, and parameters can represent most data types within the Fortran context.

Proposals for Implicit Opening

The Fortran SUBROUTINE is renownedly serviceable as a vehicle of escape. In design for structures to effect an interprogramming opening, there is some choice. Immediately there is the possibility that specific interprogramming be done with EXTERNALs tailored to particular uses. However, such private facilities have several drawbacks. As they are the concern and responsibility of the programmer, matching any connection or communication standards becomes entirely a programmer requirement and indeed responsibility. And such impositions must be borne in each particular use. Also, of course, the device is less clear in intention, not obviously an opening in context. Beyond such immediate approaches, though, a general interprogramming offering might be constructed from SUBROUTINE and FUNCTION. As discussed more generally, for instance, either "send-receive" or "request-reply" structures might be employed.

Coping with data encodings differing between programming language environments is an important interprogramming concern, but not every language is appropriate to this concern. The Fortran situation is equivocal. There are the mapping facilities involved in input/output, and Fortran practice is not shy of more general low-level manipulation. But the packaging and the transport are tightly bound in standard Fortran, and while mapping is character oriented, there is no provision for character manipulation. What facilities there are involve the "hollerith string" and the input/output "record". In fact, many Fortran languages feature non-standard facilities at this point, for example the much-followed IBM Fortran IV (IBM, 1964a) with ENCODE and DECODE - which allow format conversion directed to or from an array. Or, relying perhaps on Fortran "weak" typing, appropriate procedures with hollerith parameters might be employed, or EQUIVALENCE name overloading resorted to. None of these methods address the possibility of varying length sequences of characters very well. Hollerith strings assume otherwise available knowledge of sequence length. Some flexibility in format specification is available through use of a hollerith string in an array standing in for a FORMAT statement. But varying length structures will need formats specifically so set up in advance, and possibly preliminary investigation of the data itself before even that can be done. So some data mapping flexibility might be provided in Fortran, most probably based on hollerith strings augmented by string length numbers. Hollerith strings would also be useful as opening labels. If necessary, similar facilities might even be devised so to allow access and manipulation at the bit, rather than character, level. Such facilities might be initially offered as EXTERNAL, and if more immediate access was desired they might be built in with the rest of Fortran, and so labelled INTRINSIC.

Reliance on procedures for language opening or related manipulation is at the cost of more syntactic guards against misuse. Every parameter is a potential problem. In this language the precedent is liberal: it often allows any implementation whatever rigour it desires, while insisting on none at all.

Proposals for Explicit Opening

In opening Fortran - or any language - explicit specification is the best assurance for continuing ease in interprogramming. And incorporation of interprogramming into language syntax allows access

through all levels of elaboration. As, for example, may be argued for syntactic input/output provision, there may be recourse to any knowledge of "type" perhaps later unavailable.

The send-receive model is so closely approximated by the existing input/output it seems best simply to further extend that opening. The limitations discussed earlier must be removed. The record orientation itself is not objectionable, and character granularity is the basis for any formatting. There seems no reason why records might not vary in size, although what that size was, if it could not be assumed, would have to be dealt with explicitly. So there is no reason that such records could not themselves define the interprogramming granularity quantum. As for the limited expressiveness of a unit number, a unit name identifier should be just as functional - syntax as any other Fortran name.

The request-reply model is not already available, but there are the structurally similar possibilities. The outward opening, the "request", is similar to the CALL to a SUBROUTINE. The name subject of the CALL may already be declared not immediately present either INTRINSIC or EXTERNAL. To extend the opening of CALL, another declaration, say EXTRINSIC, might be added. The name so declared could be the opening label for outward interprogramming. The inward opening, the "reply", is similar to the SUBROUTINE that is called. In distinction from the model of "send" or "receive", a "reply" incorporates paired bounds on control. In Fortran, only a SUBROUTINE provides such bounds. To make a SUBROUTINE available, syntax might be extended to complement the CALL statement with an ANSWER statement. The name subject of the ANSWER, and of course of some SUBROUTINE, could be the opening label for inward interprogramming.

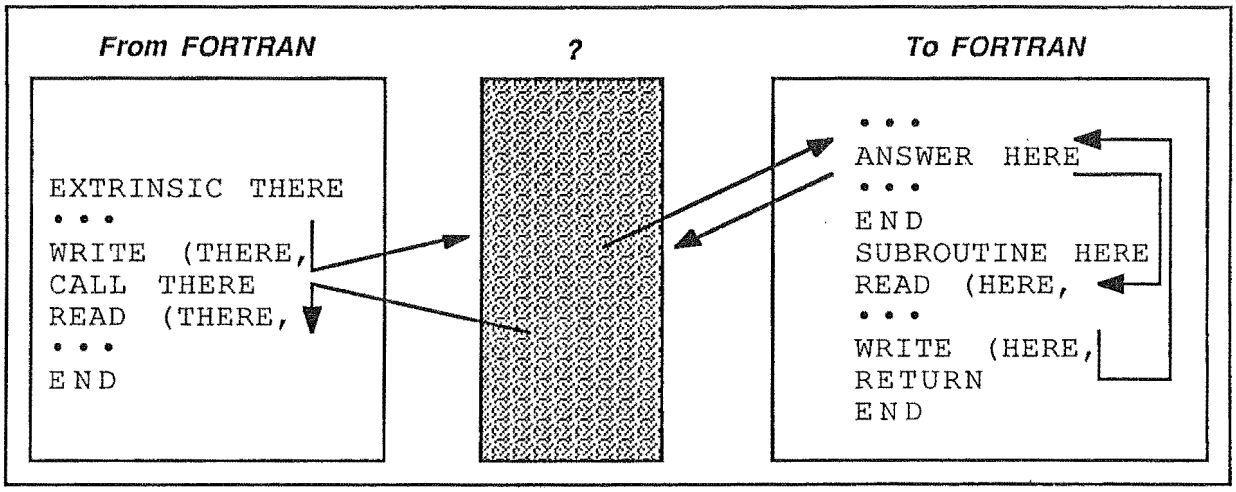

Figure 6.1.1: Fortran Interprogramming - Explicit openings using CALL-EXTRINSIC-ANSWER and associated named I/O units.

In using the SUBROUTINE to embody the opening connection, the opening communication then suggested is the SUBROUTINE "parameter list". Fortran parameter passing is defined "by name" (though not for expressions) and frequently implemented "by reference", with no distinction of direction of information transfer. As no indirect or "side" effects should be allowed or allowed for, "by reference" can be made "by copy" in interprogramming. A CALL to an EXTRINSIC SUBROUTINE might include copying

all actual parameters specified initially outward, finally inward. An ANSWER with a SUBROUTINE might include copying all formal parameters specified initially inward, finally outward.

There are two problems: the need to copy everything in and out; and the question of parameter formatting. Of course the copy problem could simply be endured, and the FORMAT problem dealt with by requiring declaration and use of any consistent format. Little more can be done to directly address either problem without some further extension. Indirectly, however, both could be helped if the (extended) facilities of input/output could be used not themselves modelling send-receive, but in assisting request-reply. With CALL and ANSWER providing the connection, READ and WRITE might provide the communication: both pairs together providing the context opening. In this way, only that necessary need be transferred, and the available data mapping facilities may be used as required. If units were specified by identifiers instead of numbers, it could even be possible to also use the EXTRINSIC names for units providing their communication.

Conclusion

The syntax of Fortran is old and not particularly easily dealt with, and many Fortran compilers are perhaps the same: best left alone. The easiest way to open Fortran is by a set of subroutines, there need be little or no other impact on either implementation or language. If access to the security of syntax is important, the use of input/output is the simplest way: some language redefinition is required, to assure about granularity for instance, but the language surface may remain. Rippling that surface, i/o unit numbers might become names. Should explicit incorporation of request-reply interprogramming seem worthwhile, SUBROUTINE usage might include EXTRINSIC and ANSWER: additions to both language definition and surface syntax. In even more daring opening involving communications directly via variables, the language must demand and the implementation must provide statement of protocols and encodings for such communication. Using READ and WRITE in conjunction with EXTRINSICs, however, would make for an easier solution.

Existing openings might coexist with new openings, or might indeed simply ignore them. For example the number used by STOP might seen as being written to some not explicitly labelled unit - or might not. Flexibility in communication and coordination on entry and exit has proven useful in other contexts. Many Fortran implementations have some local method of practice, so integration of these openings might even foster some minor convergence. To access entry "parameters" (on the Unix and C model) in a Fortran environment without any special provision, Kernighan and Plauger (1976) suggest simply preparing and reading an input unit. This strategy, and perhaps similar output on exit might be especially appropriate in an interactive environment.

Fortran brings some helpful facilities usable in the provision of interprogramming. EXTERNAL may provide an easy beginning, INTRINSIC a more blessed state, and those names and categories bases themselves for extension. SUBROUTINE and FUNCTION provide facilities on the way to interprogramming, or the interprogramming facilities themselves. And the flexibility, not to say looseness, in

parameter matching might well prove useful in less explicitly established facilities. However, some facets of Fortran make for difficulties. Data mapping facilities are limited to input/output, for instance. Parameter passing is not directional, and so both must be considered. But these are language choices, and can be tolerated.

So might there be new Fortran. Yet again. But unlike many many other Fortran extensions, this adds very little, almost nothing: while offering very much, almost everything. It offers escape from Fortran, or escape to Fortran. Many programmers wish to escape from Fortran, but are bound by the wealth of programmes upon which Fortran sleeps so sound and long. Many programmers wish to escape to Fortran from other languages to access that software, especially the numerical software. Fortran is also a language that permits programmers, on the merest pledge of honour, to indulge in the dirty horrible wicked work that programmers occasionally find themselves needing to do - defeating type checking and worse.

SECTION II. PASCAL

Pascal was originally designed with the teaching of programming in mind, stressing ideas of structure in data and control within a simple yet general frame. Though without any direct strong commercial drive or support, it has become very widely popular, particularly in environments concerned with the teaching of programming, both institutional and personal. This popularity at such a critical stage suggests Pascal an important subject for interprogramming: a base and a retreat. In this discussion, the Pascal variant of concern is probably the most popular, the amended original (Jensen and Wirth, 1976) to "level 0" of the ISO Standard (ISO, 1980, Addyman, 1981).

Existing Openings

The explicit openings of Pascal are again the two most common, entry/exit and input/output. Entry and exit are themselves simply a lexical beginning and lexical end. However, coupling both and explicit in opening is the "program heading". This facility seems intended to allow extra-programme access to variables specified "external" by inclusion in the heading, such access deemed "implementation dependent" (but compulsory) for type file, and "implementation defined" (quite optional) for anything else. In most practice, only file or text variables are affected. And while meaning may be implementation dependent, it is claimed that "An external file exists independently of any program activation", and "After appearing in the program heading, external files are declared and treated just like ordinary file types" (Cooper, 1983). Were this strictly true, Pascal might appear not to encompass any further opening for input/output.

In practice, however, file and text operations commonly escape the Pascal context directly and dynamically (sometimes without deference or even reference to the program heading). Indeed, only such practice is reasonable in interactive computing, where file input and output must dynamically reflect programme control. Seen as interprogramming openings, these operations have two drawbacks. Firstly, no externally assured granularity is guaranteed or suggested. Secondly, while flexible data mapping facilities are provided with character based "text" file operations, none at all are provided anywhere else. In practical programming it is often desirable to have facilities for at least conversion between numbers and character arrays, without any wish for immediate input or output.

The implicit openings in Pascal are subprogrammes: functions and procedures. Most within the language are procedures and functions that need not be declared, such as might be called "required" in the language of the standard but that they are not in fact required in every implementation - though are permitted in any. Less entirely within are procedures and functions declared with "directives" (most commonly "external") that locally indicate some exterior definition. Both these approaches allow for access from inside Pascal to whatever is outside Pascal with no strings attached - or provided. Control is certainly precise, and parameters may pass the large part of the Pascal world made up of values and variables. One possible problem is the strict type rules that apply in parameter matching and the limited flexibility inherent.

Proposals for Implicit Opening

A Pascal interprogramming facility should probably provide transfer of any sequence of any mixture of variables, in any granularity thereof, through named openings; data mapping facilities are desirable. Most straightforwardly, subprogrammes might be used that specifically address specific interprogramming by specific implementation, but such is a programme, and not a programming language, concern and responsibility. In using subprogrammes to more generally provide this facility for the language as a whole, a matched pair might follow the send-receive model, another matched pair (or perhaps triple) implement request-reply.

At this point arises the difficulty of designing a suitable parameter structure. The problem concerns flexibility. The flexibility needed in interprogramming variable transfer is the need to deal with variations of type and of number. The flexibility offered in the Pascal philosophy about procedures and functions seems a little unclear. Subprogrammes "declared" (regardless of directive) must include specification of formal parameters, and the matching with actual parameters in summoning calls must meet strict rules. Subprogrammes "required" and "provided" not only need not be declared - so no formal parameters are declared - but in summoning calls any parameters need not meet the strict rules for any one possible declaration. Indeed, such parameters need not even meet the otherwise applicable rules of syntax: "write" and "writeln" parameter lists have their own syntax.

For the "declared" subprogramme case, the rules seem reasonably clear. There is no provision for variability in either type or number of parameters. However, in types themselves there is some of both. A file structure is a arbitrarily sized sequence of some type, so a file parameter might stand for a sequence of data. Such files would not need to be external, but it is a drawback that implementation may often be via secondary memory and slow. There is no reason why this need be - a more flexible and so possibly more efficient scheme for "internal" files might be more generally appropriate anyway. A record structure with a variant part but no tag field is a free type union, so a parameter of such a type might stand for several different types. The main problem with so using the record type, however, is that each encompassed type within the union must be explicitly distinguished. There is no universal union on offer.

Coping with this might be attempted inside or outside Pascal's rules. Inside the rules, the parameter might be given a record type indicating a free union of a variety of required types. Making this variety wide would be difficult however, remembering that even arrays of different size constitute different types. Outside the rules, the parameter might be declared a record type indicating a free union of whatever types are required in the particular circumstances. In much the same way "write" might be seen as representing a sequence of particular procedures appropriate to whatever sequence of types of variables are in each call. The gap left open by allowing directives is vague, however, and it's difficult to know how naughty one might be and still see linguistic harmony retained. Is it reasonable to allow the same parameter to have different types in different programmes? In different declarations in the same programme? And if such is possible for the differing detail of record types, which does mean differing types, then why bother with the record masquerade at all? Or indeed the use of files? And what about access to differing procedures or

parameters? There does not seem to be any understood policy or hierarchy of tax on such sin. On the one hand, any such liberty is an admission that the entire world is not Pascal. There can be no pretense, for example, that such directive declared subprogrammes are perfectly ordinary and respectable but, alas, unavoidably detained elsewhere. On the other hand, the structural rules in question may simply be seen as a cross-checking device. And as long as subprogrammes are declared and directed appropriately, responsibility beyond the call might be seen as independently assured. Consistency in and between calls themselves can still be assured syntactically as before.

If it is decided that the parameter rules may not be broken even for directive declared procedures and functions, there is one more possibility. While the general file type allows variability in number, the required "text" file type allows that and more. Because the required subprogrammes operating with text files allow mapping from many other types, text is a type that itself encompasses both variability in number and type. So by representing interprogramming data always by text files, interprogramming subprogrammes might allow flexibility yet meet all parameter rules.

The next step is consideration of labels to distinguish openings: the practice for much similar in Pascal is of course use of identifiers. So in this implicit opening, strings holding identifiers would be appropriate. But then there is the question of the parameter rules and sizes in array types, assuming no conformant array schema. There are a few options: bless some particular size and insist on it; use integers instead - they're good enough for labels; or even use a file again. In fact, the same file might be used, label preceding data. If it is decided that the parameter rules may be broken, the approaches using "declared" and those using "required" subprogrammes are rather similar.

For the "required" subprogramme case, the rules seem few. There is provision for variability in both type or number of parameters, and syntax too. Accordingly, there is no problem in designing a variety of interprogramming facilities. However, there is some question about the impact on the language. While these facilities are called "procedures" or "functions", and while there is allowance for more than those strictly required, implementation might be considered too. Especially in compilation, the basis of many Pascal implementations, variations in syntax could not be coped with trivially. Even syntactically straightforward procedures and functions might require specific provision from within implementations to effect their addition - as "write" requires for it's special formatting syntax in its actual parameter list. At this point it becomes difficult to distinguish between language and implementation, and the difference between implicit and explicit language opening is rather blurry. The implications seem sufficiently non-trivial to move to more general discussion of explicit opening and language change.

Proposals for Explicit Opening

If the language is to be changed to realise sufficiently useful explicit opening, the modest approach seems to work with what explicit opening is already present. In Pascal, this means a look at the program heading and external variables, and particularly at the types "file" and "text".

As a first step, the explicit opening through an (external) file or (external) text might be more precisely specified. Most particularly, the communication granularity quantum needs to be defined stating what point data crosses the language boundary. In interprogramming, this is necessary in order for proper, which is to say at least determinable, coordination. For the general file type, the most reasonable quanta seems the component type of the file; little else seems conceivable. In this way, a file of char would have granularity of one character. For the required text file type, however, the "line" might be more reasonable. This would match other differences between "text" and "file of char" tailored to common applications involving document-like application objects. The view of Pascal wherein external files are not particularly an opening but merely pre-set and post-examined variables would correspond with a granularity quantum of the entire programme. This might be suitable for "batch" computing, but for interactive computing the explicit opening is necessary, and also the finer granularity. The common solution to Pascal's problem with batch oriented files coping with interactive input, now allowed in the ISO standard, has been the approach of stating that no "get" will be done for "input" until the first "readln": "lazy i/o" (from Saxe and Hisgen, 1978). The interprogramming approach suggested would go along with the suggestion that "Lazy I/O Is Not The Answer" (Perkins, 1981) and that lines could be considered distinguished by "separators". This would achieve a similarly useful solution, though in a more generally applicable way.

A next step might be to generalise the types involved. After all, file and text are only openings at all when external, and any top level variable might be declared external by inclusion in the program header. So, for instance, any external variable might denote an opening. Interprogramming labels are no problem, as with file variables, their names themselves should be sufficient. This does admittedly imply potential difficulties in parallel structures such as expressions and parameter lists where the order of evaluation is not specified by the language. Either such non definition could be eliminated (so perhaps implying implementation inefficiencies in architectures with other direction stacks), or they must be lived with as is done currently where function calls present similar problems. Generalising granularity is a bit tricky too. If the communications granularity for an external variable was that type itself, it would be difficult to justify why such was not the case with files and text s - they are variables too. However, granularity could be defined not by data type, but by operation. This would allow an interprogramming quantum appropriate whatever the particular access. And if the required subprogrammes involved with file and text (get, put, etc.) are regarded as operations, then external file, text, and any other variable can all be part of a unified strategy. So, for instance, mention of an array might involve all the array, but mention of one element might involve only that one element. The approach would require the language definition stating each operation encompassed, and setting out the communications quantum implied for each type or structure involved.

Such variable oriented interprogramming could easily follow a send-receive model. For a request-reply model, however, binding and bracketing communication as it does, some involvement of subprogrammes seems indicated.

The step to subprogrammes would allow escape to and provision for subprogrammes imagined outside Pascal. In allowing provision for outside recourse to subprogrammes inside Pascal, external variables could still be the key. If a subprogramme is called with external variables as parameters, the necessary communication would precede the call. Here the difficulty of unknown evaluation order would be

particularly difficult. Use of a single structured type would avoid the problem, however, and might seem most appropriate anyway. So neither subprogramme nor call need be distinguished at all. In allowing escape beyond Pascal, the imagined subprogramme would need to be declared so to enable the application of the strict rules about parameter matching. No block would be needed or appropriate, but a new directive could fill this hiatus. The similarity between such subprogrammes and external variables mentioned in the program heading also appears an opportunity. There doesn't seem to be a reason why both variables and subprogrammes can't be included in the heading. The requirement for variables to be "top level" could be extended to subprogrammes; so the heading even more resembles a parameter list, bar type declarations inappropriate in interprogramming anyway. There are still details, however, and the detail of how multiple parameters are communicated beyond Pascal, the encodings of parameters separately and together, would need be specified in implementation - as required for all implementation dependent and defined specifications anyway.

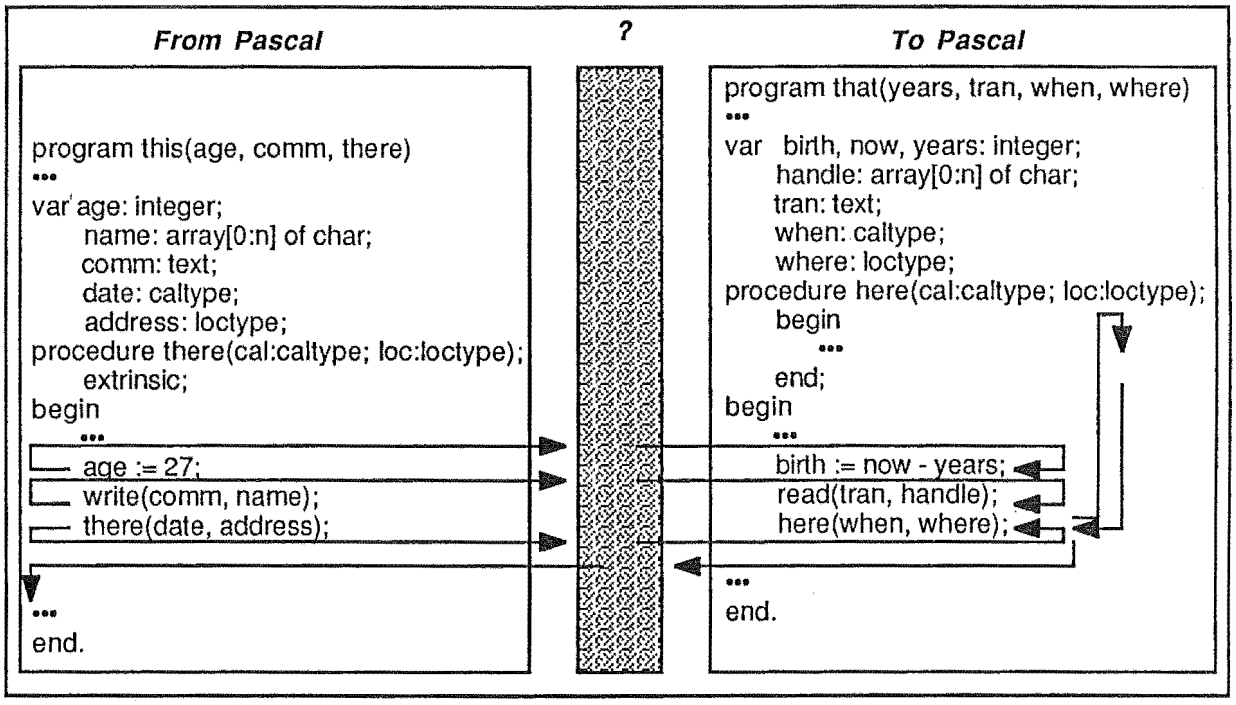

Figure 6.2.1: Pascal Interprogramming - Explicit openings using program header for variables, text files, and extrinsic procedures.

At this point some terminology should be discussed. The only subprogramme directive required in Pascal is "forward", meaning the block follows later in the programme (so enabling avoidance of circular declaration needs). The directive probably most common otherwise is "external" (or something similar), meaning the block is not in the programme, but that it will be available in programme implementation. External subprogrammes are expected to comply with expectations of any Pascal subprogramme in coordination and communication. The subprogramme interprogramming opening is similar, but more severe. The subprogramme block is not in the programme, and may be available in programme implementation, but cannot be assumed to be either Pascal or even Pascal-like , and no coordination or communication other than

that stated can be expected. Accordingly, another directive might be used, "extrinsic" for instance. Unfortunately, the closest relatives of such subprogrammes would be the variables in the program heading, known themselves as "external". One resolution would be to simply change that to "extrinsic"; it's only a language definition term after all, not used in the language itself. Another resolution might be to use only one directive anyway, say "elsewhere". If the block is given later, it would be as "forward"; if the name is in the program heading, it would be as "extrinsic"; if neither, it would be as "external". This would isolate the fact the subprogramme wasn't there, and the exact location where is was - perhaps a useful orthogonality. The documentary aspect does disappear, however, and in one-pass compilation some indirection in the implementation of the call structure would become necessary.

In all these explicit connections, the coordination is reasonably clear, but the communication somewhat less so. In data encoding and mapping, only text files are catered for. In interprogramming not using text files, the programmer is either to be left alone unless perhaps some other new facilities are designed. Of course the type freedom often extended in implementation of records with variant parts might be used, non portable and risky though it is in practice. In general communication, variables and subprogrammes included, there must be reliance on some standards which must be stated in detail as part of the implementation. This need not reflect the implementation of data within the programme, though of course this will be most efficient. In accepting data from outside, the possibility also arises of data in an unexpected format. Such an unhappy state is similar to problems that might arise within a programme, a subscript out of range, for instance. So it's probably best handled in the same way: checked for consistency if the overhead seems bearable, and assumed correct otherwise - for the brave, the lucky, the perfect.

Conclusion

The easiest way to open Pascal for interprogramming would be to depend on "external" subprogrammes, should that directive be available. But should it not, and in general it is not, more explicit provision must be made. In that same orientation then, the necessary subprogrammes could simply be "required". More simply and consistent with the existing design, though, file control could be more precisely defined. And generalising from provision for files, other variables and subprogrammes too might be integrated in opening. Of course, it is necessary that the language insist that an implementation provide the necessary communications encodings and protocols for any necessary connection. And it is necessary that the implementation be documented stating precisely what these are, as with all implementation details.

In looking to more completely integrate some interprogramming facilities into Pascal, some existing facilities appear quite helpful. The program heading is reminiscent of a subprogramme heading, and might be seen as an opportunity to express the involvement of the programme within some exterior environment by means of parameters. In this way the opening used for external files is not only made explicit, but available for generalisation - made more clear in that files themselves are a type of variable. The provision for replacement of the block of a subprogramme by a directive also might seem to admit the

environment not Pascal, even though no directive that directly says anything like that is actually required.

Some aspects of Pascal, however, do not look helpful. One problem is simply coming to terms with some aspects of existing Pascal design that do not themselves appear harmonious. The provisions for input/output do seem oriented about "batch" processing, and the demands of interactive programming are met only by the blatant of "lazy i/o". And further with input/output, the apparent philosophy that the operations involving files are simply subprogrammes seems unconvincing when they are so privileged in syntax that understanding of subprogrammes themselves seems threatened - and in a learner's language. Another kind of problem arises in attempting to make use of the opportunities present in program heading and subprogramme directives because of the local indeterminacy in evaluation order of parameters. This can be coped with, but the allowance of indeterminacy may be less help in implementation than hindrance in extension. The lack of directionality implicit in "var" parameters also is a nuisance in communicating between programming languages. Other problems in interprogramming practice may simply derive from the strictness of data type outlook. More facilities for mapping between types might be useful, are not provided by the language, and are difficult to construct within the language.

A central concern in Pascal is rigour within flexibility. Practical computing needs insist, however, that a practical language be at least somewhat open. Extension of existing openings for interprogramming need not detract from rigour any more than is already the case. And by so doing, Pascal might allow access within to the clear and simple context in which much model software is presented and with which so many even new programmers are familiar. And by so doing, Pascal might allow access outside to old programmes in old languages, special purpose programmes in special purpose languages, or simply filthy programmes in simply filthy languages.

SECTION III. C

C is primarily a language for implementing "system" software, and has become popular and widespread with the operating system it was created to support, Unix. This alone is enough to compel interest in C interprogramming. In fact interprogramming should concern any software base language for the important access it affords to environment of implementation. Like most system programming languages, C has an orientation about efficiency of implementation, precision control and minimal overhead. Any interprogramming should probably respect such values. In this discussion, the language of concern will be that described in the "de facto" standard, Kernighan and Ritchie (1977), including the standard "Stdio" input/output library, but excluding any assumptions of a Unix environment.

Existing Openings

The common entry/exit and input/output openings of C both simply rely on the "function" structure. As such, neither is explicit in syntax, though some common input and output functions involve dubious variability in number and type of arguments. Entry to and exit from a C programme consists of entry to and exit from a function distinguished by the name "main". The "main" function has two parameters specifying an array of character arrays, though declaration or use of these parameters is optional. No "return" value of "main" is assumed or required; should one be provided it would by default be regarded an integer. Conventionally the parameters describe an exterior "command line", separated into textual arguments, that precedes the programme. Conventionally either no value is returned, or an integer value signifying either "success" or "failure". With some flexible character translation facilities available, entry to a C programme seems of use in interprogramming. With some structuring of any separate parts into a single whole, and again perhaps with suitable character translation, exit from C seems useful too.

Input/output is not strictly part of the C language at all, but the "Standard I/O Library" is very widespread and often may be specially catered for in implementation. The facility is oriented about communication via "files" with the external environment. Files are assumed to be externally distinguishable by a character array as a name, though internally this name need only be used in initial specification and association with a structure type "FILE". Actual communication can be precisely controlled, there being not only methods for input and output, but also for specifying transfer quanta ("fflush"). There is good flexibility in data translation, not only on input and output, but also to and from character arrays - all done with functions. All this seems clearly well-suited for interprogramming. It is, however, only the send-receive model.

Proposals for Implicit Opening

So it then seems reasonable to consider how a request-reply model might be provided. Moreover, as the entry/exit opening is in a sense already a limited request-reply pair, it would perhaps be pleasing to integrate the different openings. As all the existing language openings depend on functions, continuing to

employ such implicit openings seems the only appropriate. To build on the input/output opening, only the paths in and out of C that are "files" could be used. Files can only be used in one direction at a time, so this makes for some restrictions.

In the "request" case, data from C is to be communicated out, then data to C is to be communicated in. These two components are in request sequential, and there is no reason why "send" functions such as "fwrite" or "fprintf" (with formatting) cannot be used followed immediately by use of "receive" functions such as "fread" or "fscanf" (with formatting). If a united facility is desirable for documentary reasons, a function combining both parts could be offered. An "fcall", for instance, could take as arguments "FILE" pointers for outward and inward communications, with buffers and sizes to match. Buffers of "char" could be constructed before request using "sprintf", analysed after request using "sscanf", both allowing formatting as if to a file, but in fact to a character buffer. Or the translations could be done by other specific functions, even perhaps nested themselves as buffer parameters. Though less appealing because of the problems of two open-ended lists, a specific formated version, say "frequestf", might be offered too.

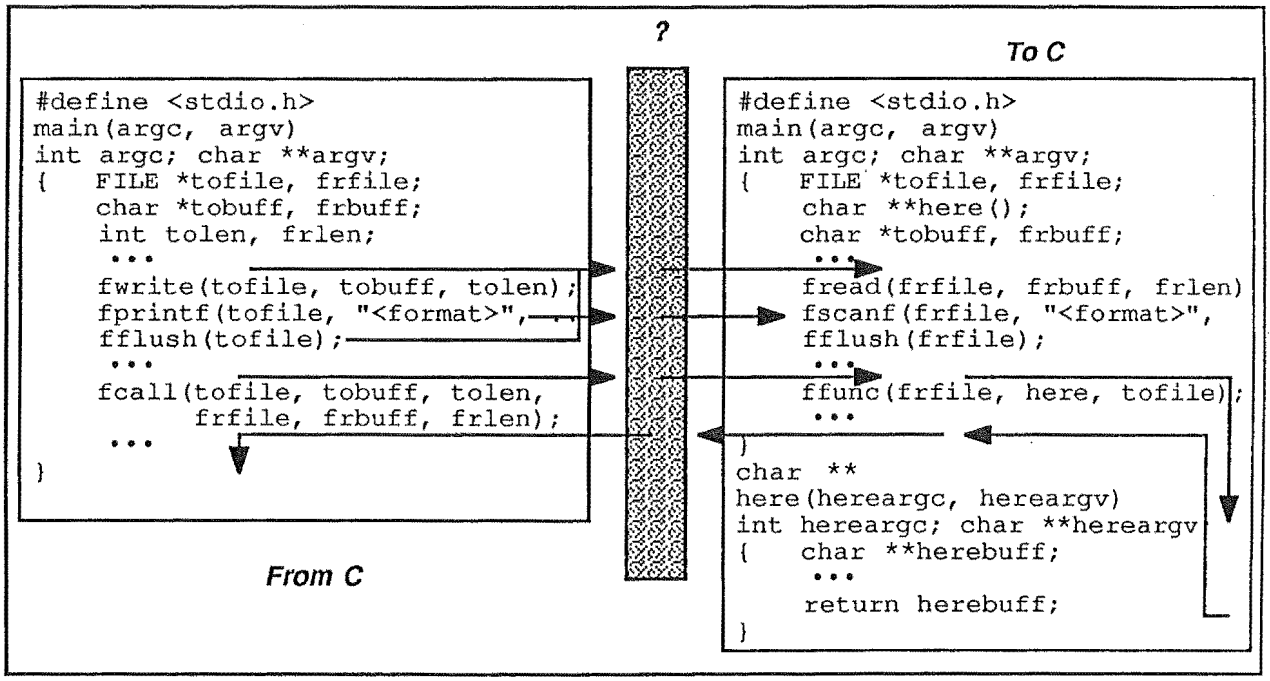

Figure 6.3.1: C Interprogramming - Implicit openings using Standard I/O for send-receive, and proposed functions "fcall" and "ffunc" for request-reply.

In the "reply" case, data to C is to be communicated in, then C must do as is to be done, then data from C is to be communicated out. These two components are in reply separated, yet must be bound as brackets about that between. Especially in this function oriented approach, that bracketing suggests a function for the processing between. And of course another function to bind the reply together. That function, "ffunc" say, could take as arguments "FILE" pointers for inward and outward communication, and the function to be called between them. That reply function could get the inward data as parameters, and put the outward data via "return". So "ffunc" could also take as an argument one other function to take care of the

needs. And that argument could be optional, in which case a default function could be provided. The default function, say "fmainf", could use the conventions of the function "main", but, say, returning a buffer of strings. This would make for symmetry and integration among C entry/exit, input/output, and request-reply. The "FILE" pointers passed to "ffunc" might be made available to the reply function externally as say, "funcin" and "funcout". These could be available to "main" too, and perhaps more for permanent references to parallel the "stdin" and "stdout" provided by the standard library, they could be called, say "mainin" and "mainout" .

Proposals for Explicit Opening

If these openings dependent upon the flexibility of the C functions were not considered enough, a facility to explicitly open C might still be built into the language. The "storage class" attribute set might be extended from "auto", "static", and "extern", to also include, say, "extrinsic". And "extrinsic" items would be as parameterless functions usable inside C, but in communication beyond C. There would be many problems and details to be resolved. For instance, there are the problems of granularity again, and of unspecified evaluation order again. While these might be resolvable with language change, the change seems too severe. Even in the most common language openings, C does not hint at such explicit recognition: all is done implicitly via the function. And knowing C is most commonly used in situations where efficiency is important, and where language size may well be significant, any large effort simply to make opening explicit would be of doubtful worth.

Conclusion

C is easily opened for interprogramming. The important reasons for that ease are the flexibility of the the "function", and, using that, the standard input/output library with its very flexible but still precise control on offer. There are some drawbacks, too, though. That same dependence on the function means some cross checking about type and number is difficult in C. The syntax checker "Lint" (Johnson, 1978) can detect some problems in a static C programme (and must be informed of rule bending "standard" functions), but cannot detect indiscretions dynamically. In addition, there is some lack of symmetry in function parameterisation: only one value may be explicitly returned, even if it is structured as is allowed in some later versions of C.

As a language of and for implementation, C has much to offer through interprogramming. When any otherwise unconcerned language needs access to some facility at the lower level, C may well be the right partner. When C itself, in implementing new software, might build upon already established software at a higher level, then that service might be returned. Even where there are strong efficiency concerns, often the case in systems implementation, interprogramming might facilitate ease in prototype development and experience. C itself is in some environments simply a well supported general purpose programming language, and so even more general demands might be expected. The simplicity and flexibility of C with standard input/output library should well support all.

SECTION IV. ICON

Icon is a descendent of Snobol, and in that tradition is intended primarily for non-numeric applications, particularly those involving complicated character based textual processing. In this application area, Icon allows very brief specification of programmes that would in most other programming languages be very extensive. However, while general programming in Icon is straightforward, it isn't intended as the linguistic answer to all questions. Moreover, the language does place some requirements for overhead on an implementation that may seem unacceptable to some applications, or at least some parts of some applications. Icon is normally implemented by interpretation, and though compilers do exist, they too must bear some interpretive overhead. So Icon is clearly of interest in interprogramming: inward it may afford a facility for text processing otherwise difficult; outward it might lean on languages with other specific orientations, or on languages more efficiently implementable.

The outer visible structure of Icon, unlike Snobol, is in the Algol tradition, most resembling C (via Ratfor, an early implementation base). But it is not C, and both offers and requires facilities C does not. In looking at interprogramming with Icon, the parallels with C interprogramming should be illustrative of the generality sought.

The particular Icon language here considered follows "Version 5", described by Griswold and Griswold (1983), but does not assume a Unix environment, and so still much concerns the more widely portable earlier versions.

Existing Openings

Like C, Icon has little in the way of explicit openings, and the implicit openings used resemble those of C. The structure of an Icon programme is a set of functions, one of which by the name "main" is distinguished as entry and exit. While Icon functions may have parameters, no particular parameters are required or assumed of "main"; while Icon functions may return values, no particular return value is required or assumed of "main". So the entry/exit opening is clear in coordination, though less so in communication, but the support potential is there.

The input/output opening is via a set of functions concerned with "files". These are assumed to have external names expressible as character strings. Functions are provided to "open", binding an external name with an internal one of type "file", to write to files, and to read from files. All communications is in terms of characters. No assurance of communications granularity is provided, although the facilities do imagine files being of "lines" of characters. As for data translation, of course, given the character orientation, Icon provides excellent facilities of great flexibility. So as long as something can be done about the communications granularity, Icon input/output is certainly suitable for interprogramming, on the send-receive model.

The implicit function opening is again quite flexible in Icon. As values, rather than variables, are typed in Icon, there is flexibility possible in argument type - indeed it can be determined dynamically (via the

"type" function). The number of arguments is flexible too, as arguments expected but not supplied simply default to the (detectable) null value. Similar flexibility is available for the "return" value, though it must be a single value. Some functions are pre-defined in Icon, an opening for more. Other functions may be declared "external", suggesting the possibility of depending on common implementation bases, though various conventions must be adhered to, and of course no protection is offered. This does offer the possibility of offering new facilities in Icon without changing the implementation at all.

Proposals for Implicit Opening

As Icon input/output is otherwise suitable, the first step in interprogramming should probably be the specification of the granularity of communication. While many of the functions provided are concerned with "lines", some are not. Perhaps the best course would be to assure granularity on a function call basis. In a language where efficiency in implementation was a main concern such an input/output policy might be unacceptable, but it should not be a problem in this case. Anyway, it might be possible to assure such granularity where it could be of significance, and behave more efficiently where it couldn't be.

In considering offering a request-reply modelled interprogramming facility, the Icon function flexibility would prove useful. Icon files are directional, so two are required; they take and yield only strings, so two strings are required. A "call" needs simply to take the two files and outward string as arguments, and return the inward string as a result. An "answer" could take as arguments inward and outward files, and a reply function to bind the brackets in processing. The reply function itself could take an inward string as an argument, and return the outward string as a result. Data translation seems so much a part of Icon that nothing more need be done, but of course one more level of function as argument could easily be added if translation were there desirable.

While Icon might at first seem fairly conventional in its function structure, the unconventional "generation" facility is at the heart of the language. A function may generate a value by using "suspend" instead of "return". An expression incorporating such a value is itself evaluated iteratively, each time "resuming" any generation after the "suspend" point. This is in fact another coordination model. Whereas send-receive is one communication in one direction, and request-reply is one communication in each of two directions, generation is one communication in one direction and many communications in the other. Any language might wish to model such a structure, of course, and may well by using whatever is at its disposal. For Icon itself, however, such an interprogramming model might seem more obvious. In providing the inside, a reply function need only suspend as well as returning. The answer function must then accept resumption cues from outside, resuming the reply function accordingly. In structuring a view of the outside, the call function might imagine suspension beyond icon, and resumption from within. To behave as an Icon generator, inward communication would need to be identifiable as "suspend" or "return", and "call" could then follow suit.

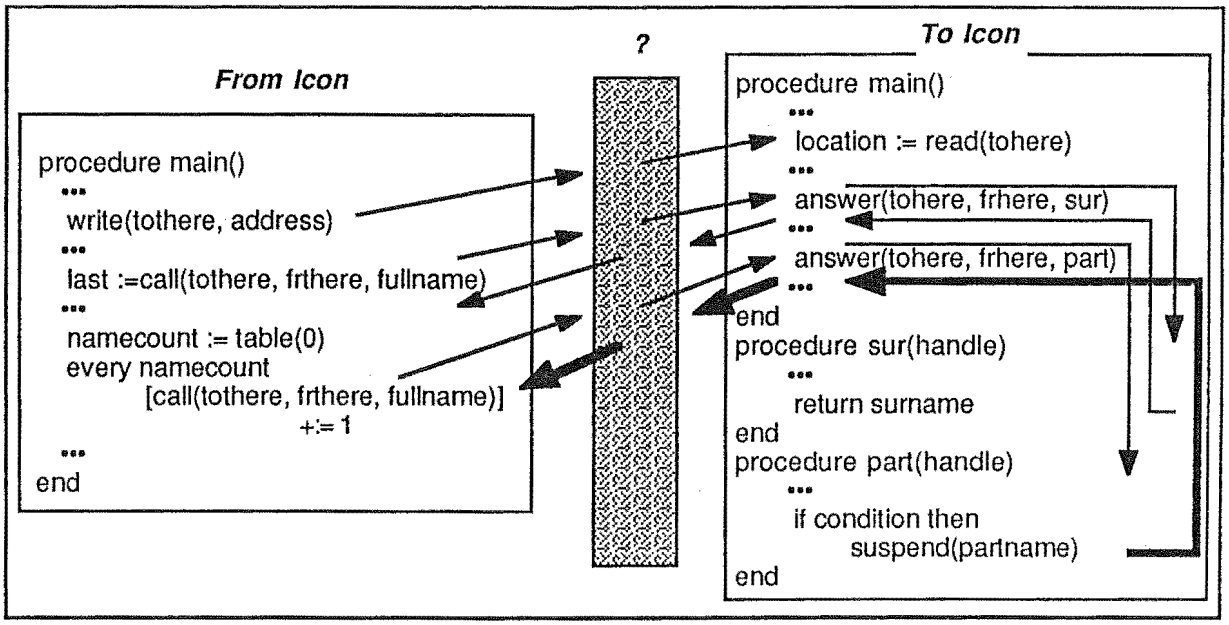

Figure 6.4.1: Icon Interprogramming - Implcit openings using write and read for send-receive, and new functions call and answer for request-reply. Calls generated by "every" result in potentially many results from suspension of "part".

Proposals for Explicit Opening

The closest Icon comes to an explicit opening now is in the "external" declaration for functions allowed in the compiler implementations.

An extrinsic function could in operation behave like the "call" function described earlier. The function name itself could be used as a basis for interprogramming labels, and any provided arguments could be converted to an outward string, with the inward string returned - possibly with a sequence of strings from suspension. The matching "answer" facility could conceivably be made part of the language syntax, but many other important parts of Icon are simply predefined functions, so it seems an extreme step. The lack of symmetry might seem disconcerting, so perhaps it's best to leave the syntax alone. After all, the input/output facility also manages without explicit language support.

Were the function oriented interprogramming facility available, one syntactic change that might be considered is the allowance of "external" in interpreter as well as compiler implementations. In compiler implementations it might work as it does now, retaining the advantages of efficiency. In interpreter implementations it might work as the "call" function, offering outside access and implementation compatibility. The "call" function itself might still be offered in both implementations for situations where the more general and protected access though interprogramming is indicated.

Conclusion

The general structure of Icon is like that of C, and interprogramming might be offered in a similar way as discussed. But Icon is quite different from C in orientation about generation, even though it can be handled retaining the same structure. And Icon is quite different from C in implementation, depending on at least some interpretation. So though similar, an interprogramming facility is needed from Icon to C where the implementation is too interpretative to allow external C functions, and is needed from C to Icon anyway.

The interpretation afforded flexibility in typing means that interprogramming is easier to add to Icon without flouting any rules. Also, the nature of the language itself affords flexible data translation facilities very useful in interprogramming. The specified input/output facilities are oriented about characters within files without any intermediate step so not allowing precise control over communications granularity. However, some use of either function call or "newline" characters might allow soft control.

Icon is be attractive because the attention given character string processing is rare. Interprogramming through Icon seems reasonably straightforward and should allow potentially very useful interaction between this context and those with other priorities.

SECTION V. PROLOG

Prolog is a language for programming based on the logic of predicate calculus. It is very simple in fundamental structure, sufficiently so to ease formal tractability. Yet it is a quite capable and potentially useful language, especially for programming applications concerned with complex yet declarable logic. As a result, Prolog has received much attention in artificial intelligence research otherwise typically oriented about Lisp. Despite the importance attached to the tractability of the set of declarative semantics of the logical base, there is another parallel set of operational semantics as well. And the operational semantics constrain the language making practical programming a possibility.

The language is little other than logic, though, and otherwise addresses no directly practical application. Moreover, the language is mostly or largely implemented by interpretation, adding the burden of interpretive inefficiency and a rather closed environment in further dissuading direct practical programming. So for both the promise of logic programming, and this payment for it, some interprogramming strategy seems of interest.

In this discussion of Prolog the language of concern is roughly that described by Coelho, Cotta, and Pereira (1980) or Clocksin and Mellish (1981) and the documentation of CProlog (Pereira 1983) excepting any Unix assumptions. The language conventional syntax has changed since the original Marseille version, and has no "standard", but the simplicity of the fundamental structure remains.

Existing Openings

A Prolog programme is a very regular structure, consisting of a set of procedures consisting of a sequence of clauses consisting of a procedure heading and a sequence of goals themselves procedure calls. The operation of the programme is the unification of procedure headings with procedure body goals binding otherwise universally quantified variables ultimately upon fixed atomic values. Even though some special syntax is provided, for lists and for arithmetic, for example, it is offered only as surface convenience and is defined on the more general structure.

So there are no explicit openings in the syntax. Implicitly, however, there is the procedure of course, and a number of pre-defined procedures are provided. There is no particular consideration for entry/exit, but input/output is provided by a set of pre-defined procedures. The input/output is "stream" oriented, and streams are first associated with some name from beyond Prolog and a direction inward or outward. Input streams are opened and closed by "see" and "seen", output streams are opened and closed by "tell" and "told"; "see" and "tell" take a stream name argument. Streams are character oriented, and the only inward granularity control is by delimiter characters. Some versions allow outward granularity control by a "flush" procedure, others leave the matter unspecified. Procedures "read" and "write" are available for transferring arbitrary Prolog terms, with great flexibility possible in that programme-data ambiguity, and others are available for transferring textual characters. Facilities for data mapping are available, surprisingly offering, for instance, translation between characters and an imagined integer representation. Few higher level data structures are specifically catered for, but considering the flexibility inherent in the language much

more could be constructed. So on the send-receive model, with granularity control tightened a bit, the input/output facility could well found useful interprogramming.

Proposals for Implicit Opening

In extending the facility to model a request-reply facility, a similar approach to input/output seems indicated - there is little else appropriate to follow. So pre-definable procedures to offer request and reply might be offered. For labelling outside Prolog, the same conventions in use with streams could apply, atomic names, single quoted to avoid any special characters. In a simple facility, communications might be restricted to Prolog terms - potentially most useful in interprogramming. So "trequest" might be a predefined procedure of "arity" three with arguments for label name, outward and inward terms. And "treply" might be of arity two with an argument for name, and an argument for the name of another arity two procedure to connect the inward and outward terms that will be its arguments.

In a more general request-reply facility, communications might simply still use streams, though in a bracketed structure. So "srequest" might be a predefined procedure of arity one taking simply a name as an argument, the current output stream closed and communicated outward, and a new current input stream opened for communication inward. A procedure "untold" could be added to close the current output stream, and start capturing the current output. Output captured in this way would be used at a later time, at a "tell" directive that designates a file for output, or in this case at a future "srequest". The matching "sreply" would be of arity two with a label name and a procedure to be called with input and output streams as arguments. This procedure could then do whatever is required, using "read" to get information, and "write" to return it. The "told" directive to close the current output would conclude the reply.

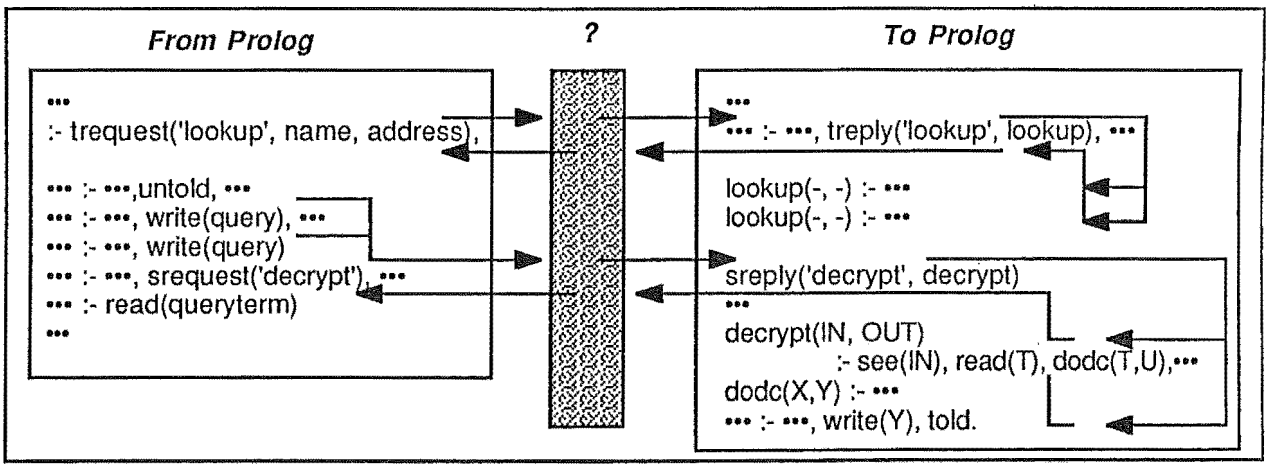

Figure 6.5.1: Prolog Interprogramming - Implicit openings using new "built-in" evaluable predicates trequest-treply using individual terms, or srequest-sreply using streams. Stream case also uses "untold" to save writes until next tell or srequest.

Central to the operational semantics of a Prolog programme is "backtracking" as matching procedures with goals is sought through the unification process. Accordingly, the procedures involving

interprogramming must be carefully used with understanding about any backtracking that might be involved. This is not difficult, and of course applies to input/output anyway, and is simply part of practical programming in the language.

At the beginning of a Prolog programme, the current input and output streams are some default, typically specifying interactive input and output. If an entire Prolog programme was considered the result of specification of some arity zero procedure specified by some "sreply" facility, the initial streams would be communication inward and outward beyond Prolog, and so offer an entry-exit convention.

Proposals for Explicit Opening

With the basis of Prolog being the logic of predicate calculus, there really doesn't seem much possibility for easily integrating any opening in context at a fundamental level. Even some pre-defined procedures already seem rather anomalous in the context of logic, truth value being supposedly permanent. However, such procedures are really all that can be tidyly done. It is that facility itself that is the (implicit) only opening, incorporating input/output, arithmetic, and the time and date. It seems enough.

Conclusion

Prolog has enough facilities, with again the proviso about communications granularity, for useful interprogramming already. With some additional pre-defined facilities - or even some constructed on that already provided - the ease of useful interprogramming should be increased further. By using a structure founded on notation already used in working with logic, Prolog offers a different context for programming than most other languages. Recourse to such notation might prove useful when using other languages to take advantage of the orientation about logic; escape from Prolog to more operationally oriented languages might seem appropriate in situations where logic must pivot on more operational matters.

SECTION VI. QUASI PROGRAMMING LANGUAGES

Thus far in discussing context opening for interprogramming, interest has been in access to and from recognised programming languages. However, the principles of interprogramming rely on a quite weak basis, and in fact might be seen to apply to more general software. Perhaps the idea of "programming language" itself might bend. The following discussion looks at a couple of common types of software, a document formatter and a text editor. While both the examples chosen are members of well known families, they are neither famous members of those families, so the more leading to exploration of the interprogramming approach on a wider basis.

Document Formatting

As a first example, consider document formatting software: specifically Doris (Peck, 1982), a portable descendent of the Unix document formatter Nroff (Ossanna, 1976, Kernighan, Lesk, and Ossanna, 1978). [So of lineage Kernighan and Plauger (1976) detail as through Roff (McIlroy, 1972) back to Runoff (Saltzer, 1965).] Doris formats in accordance with instruction lines embedded within, though not otherwise part of, a document. It is used by first preparing such a "script", then applying a Doris implementation, thereby yielding formatted "text". In this way, Doris is not really too far from a programming language: the script might be regarded as a Doris "programme", and the formatted text the result of elaboration. When a user becomes sufficiently satisfied with the final text, the "programme" may be discarded - but may also be kept for later modification or "debugging".

A simplest approach to Doris interprogramming might see the entire script to text transformation as an entry/exit facility that can be expected to consume script and produce text. Another context could then feed script and collect text - this is the relation between some Unix "mail" services and a simple formatter "fmt". While this very useful, there is further to go.

A more detailed approach might regard Doris more as a full partner in interprogramming. Both access from, as well as access to, could be appropriate. And both with some control. If Doris were considered a programming language, the one opening seen existing might be the command ".sou" (source) which inserts at the specified line script from a "file". A structure of interest in integrating further opening is ".def" (define) which defines a macro with a name, a parameter specifying character, and a following sequence of script bounded by ".."; a macro may be expanded by ".name", with the name of the macro, followed by any arguments. So ".sou" might mimic the macro practice taking a parameter specifying character and arguments. Any inward script might have granularity controlled by assuming ".." as a separator; it could nest, eliminating the otherwise potentially sticky problem resulting from other such terminators in script itself. A matching ".des" (destination) command might be offered, temporarily redirecting output text to a file until a separating ".." with output granularity by say, line - an important script orientation anyway. Together, these create a reasonable input/output opening based on the label of a "file

name" and the send-receive model. Only limited data translation is possible in Doris, especially remembering the macro facility, and no more really seems appropriate.

On a request-reply model, an additional bracketed pair of commands might be useful. A ".req" (request) could take outward and inward file names, redirecting text output to the outward name, then inserting script from the inward name. A ".rep" (reply) could take inward and outward name, and parameters specifying character and arguments, redirecting text to the outward name, including script inserted from the inward name. An entire Doris "programme" could be thought of as script following an exterior ".rep", perhaps it could specify inward and outward files, and, why not, a parameter specifying character and arguments. So yielding pleasant symmetry and an entry/exit opening.

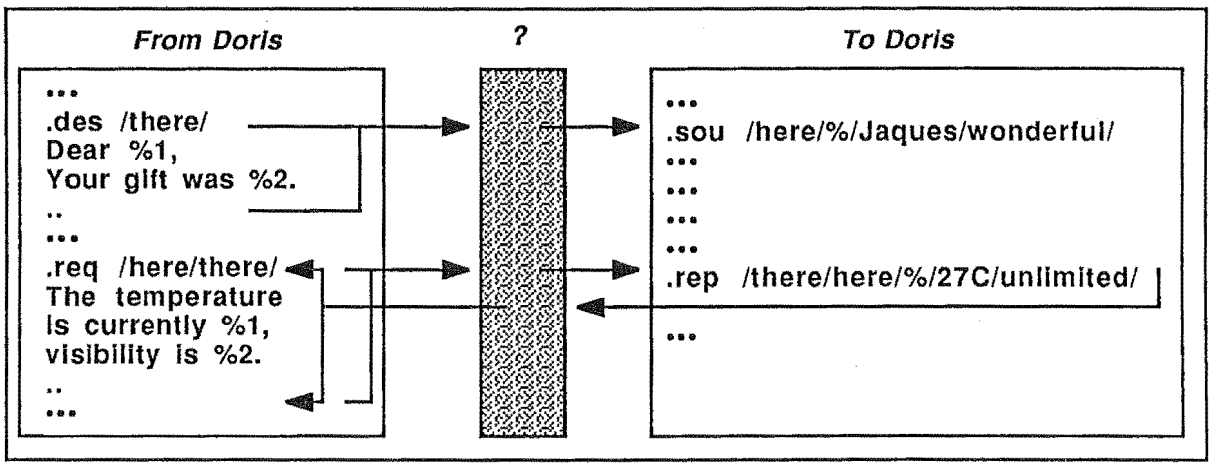

Figure 6.6.1: Doris Interprogramming - Explicit openings using extended ".sou" and new ".des" command for send-receive; new commands ".req" and ".rep" for request-reply.

At this point Doris escape to other contexts is a possibility, as is other contexts escape to Doris. But Doris is obviously limited in control, so perhaps a further step is suggested. While the purpose of macros was script expansion and argument substitution, they may be thought of too as script "variables". Assignment is accomplished by ".def", and evaluation by expansion. So on this basis simple selection and iteration commands might be considered, say ".ife" (if equal) ".ifn" (if not equal), and ".whe" (while equal) ".whn" (while not equal), each controlling a following sequence of script. These are not strictly for interprogramming of course, but are suggested by the possibilities it raises. These changes are almost entirely additive modifications to Doris - only delimiter nesting possibilities are a change - and one unlikely to present problems. So ordinary Doris could still be used for ordinary document formatting.

Document formatters offer easy access to facilities seldom offered in other software environments - particularly in programming languages. Yet the need for these facilities does not occur strictly in isolation. Well might a programme or database system wish to communicate in formatted text, and well might it wish full and perhaps flexible control over that formatting. Conversely, Doris cannot reasonably include everything (current date? current date in Roman numerals? current Roman date?) contemplatable as potentially useful in document formatting; it restricts itself largely to producing input formatted but otherwise

unprocessed. Yet as pointed out, not all Doris script is discarded on production of correct output, and in these cases might access beyond, but through, Doris lessen the effort of respecifying consequent script.

There is a diversity problem in document formatters too, and commitment and evolution with which to cope. The situation is familiar, really very similar to that of more general programming language, and the interprogramming approach appears potentially useful.

Text Editing

As a second example, consider software even further from programming language: an interactive text editor, specifically Chef (Maclean and Peck, 1982), a portable descendent of the Unix editor Ed. [So of lineage Kernighan and Plauger detail as through QED (Deutsch and Lampson, 1967) to the early Teco for the PDP-1.] Chef is primarily line oriented in its view of text, seeing anything larger as constituted of lines, and anything smaller constituting some line, with "character" atomic as a base. Single lines of text may be accommodated by a small number of "control" locations. Sequences of lines may be accommodated in "workspaces", a large number of which are available but as a stack with top access only. Chef is mainly intended for interactive use and control is directed by terse "command" lines. Chef commands, together with relevant textual data form what might be called a "session", and to stretch the terminology, a "programme". Even less than in the case of Doris are things seen this way at all, of course, and a Chef session is normally regarded as an immediate and fleeting interactive way of bashing some text about. There might be a simple connection of another programming language with Chef, as for example many interpretive languages have in timesharing implementations with similar editors to provide interactive editing in breaks from the programming language environment. But there is more that could be done.

The existing openings in Chef are several commands which involve "files", named sequences of lines understood to be exterior to Chef. The file commands offer a variety of possibilities for text transfer between a Chef session and a file, though each command only involves a single direction. In every case the granularity of transfer is quite clear. Each file command is a complete file transaction; there is no "opening" and later "closing". Where a file is a destination, the transmission is specified as an workspace, some explicit sequence of lines, or a control. Where a file is a source, the transmission is specified as the entire file or some explicit subsequence.

With this structure, Chef is quite suitable for both inward and outward interprogramming on the send-receive model. Really all that is necessary to adjust is the notion that files are exterior sequences of lines. Little difference is made as files are thought of, rather, as sources of and destinations for, sequences of lines. Such a distinction can accommodate the most common and straightforward commands and, for instance, the "EB" (edit buffer work file) command which may presume quite a different exterior file format (for efficient access), but does result in a workspace sequence of lines.

While little in Chef follows a request-reply model, something similar could be seen with some explicit addition. Probably the closest suggestion of "request" in Chef is the "X" (execute) command that

executes Chef commands from various sources and in various ways. For instance, the "XF" (execute file) command simply takes the indicated file as a part of the session, executing the lines as commands appropriately. In an interprogramming "request", some data could be dispatched outward somewhere, and some then expected inward. So lines might be indicated the outward component, and replaced by an inward component, and a "file" label might facilitate connection. This is somewhat reminiscent of the "C" (change) command, so the "request" command might be "XC" (execute and change) or indeed "CX" (change with execute). In either case, the parameters would be lines, a control or a workspace range, and a "file" name.

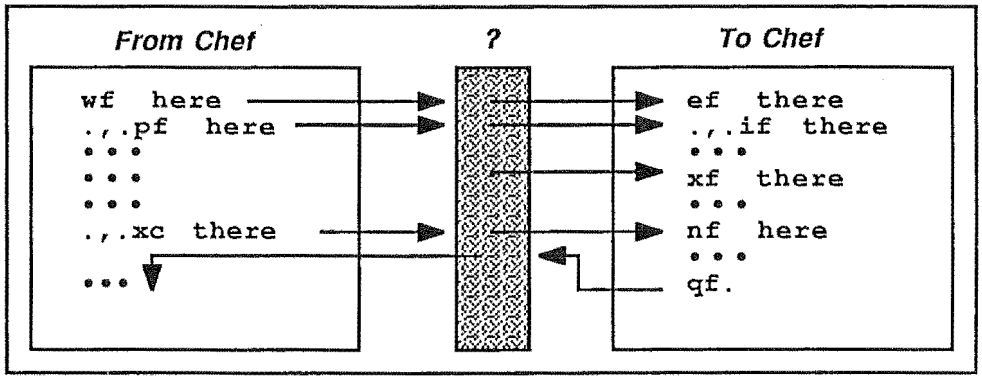

Figure 6.6.2: Chef Interprogramming - Explicit openings using the existing "wf", "pf" and "ef", "if", "xf" for send-receive; and the existing "nf" but new "xc" and "qf" for request-reply.

The closest Chef suggests a "reply" is in the file commands that fill an entire workspace with an entire file. The "EF" (edit file) and "NF" (create a new workspace and edit file) both note the file name used, the "current file", thereafter used should the character "." be given as a file name. So if following an "EF" or "NF", the command "WF" (write file) occurs as "WF.", and a suitably bracketed "reply" structure might be seen. Were a new variant added to the "Q" (quit workspace) command, say "QF" (quit workspace after writing file), then the bracketing could be made more evident. In some local implementations of Chef, a session has two (optional) parameters. The first parameter is taken as a file name, and an "NF" command specifying this file may be imagined as preceding the session. The second parameter is also taken as a file name, and following the "NF" command, an "XF" command specifying this file then may be imagined as the implicit beginning of the session. Were the "NF" and "EF" commands extended themselves to take both file parameters, and were "QF" added, the symmetry of "reply" would be completed. There do remain file commands that involve only a single direction, ones like "CF" (change from file) and "IF" (insert from file) that do not change the "current file". The distinction is that these commands affect only part of the workspace. If a "reply" structure that involves only part of the workspace seems useful, some new command or command variant would appear needed, and none seems particularly easily integrated.

In interprogramming use, some mere control power than a simple sequence of command lines might become useful. Chef already does allow for some iteration and selection through recursively expansion of controls and the "K" (kill) command. This facility is surprisingly powerful, but if more perhaps easily

understood control seems desirable more explicit control commands, perhaps variants of "X", might be worth consideration.

As admitted earlier, Chef, as most text editors, is usually regarded as an immediate tool: a session is a verb and not a noun. And it may seem difficult to imagine the need, in a Chef session, to dynamically skip elsewhere to manipulate some text. However, situations might well arise. Chef, for example, has some document formatting facilities built in: need these be so well integrated with Doris just around the corner? Other facilities not built in are similarly just as useful: word counts, spelling checking - programme compilation? There is a need for outward access, just as so with programming languages. And inward access? Text editing is a very general need. Often might a programme in a general language wish to contort some text - even a string - and have only very clumsy tools available itself. And new facilities themselves might be considered by re-packaging Chef's abilities: there is not a real need for many scanners, text editors, and stream editors to differ: they might all be one.

What is a Progamming Language?

So in fact, as for language, it seems, so for quasi language. But while in principle interprogramming facilities might be of use, in practice they might sometimes seem silly. Both Doris and Chef are command oriented in some way, and seeing them as language is not too too difficult. Collaboration between such software might be very useful, too. Text editors and formatters as discussed, but also database management systems, expert systems, and many others. Similar facilities do sometimes exist currently in some environments, but in ways restricted to particular support environments or in particular capability.

But interprogramming access cannot but have effect on design, especially integrated interprogramming design. So, for instance, the more interactivity oriented design in such software does not seem to integrate well with other software, interprogramming or no. And views even less "artefact" oriented than those discussed do exist - menu driven or pointer oriented software, for example. Such is diversity. Interprogramming is a solution to the technical implications, but the linguistic issues remain with no universal context for their analysis. All that can be done is consideration of actual interprogramming in the language design, consideration of where openings are made, and how they may be used.

In software oriented about interactivity, it seems perhaps best to regard interactivity provision a separate concern, and to offer interprogramming at a lower level. This is really no different from concerns in software conventionally thought of as programming language. In both cases there remains the questions of level and degree in opening. Should one be able to WRITE a Fortran SUBROUTINE? Should one be able to pass a "while" statement to a Pascal procedure? Such questions return to the heart of the language quality problem, and the lack of decision possible there means there can only be local decision here. Are all programmes languages? All kinds of programmes are subject to the problems of diversity, and to all kinds of programmes, interprogramming offers a technical approach to solution. There seems a continuum through various software from that conventionally thought of as programming language to where the thought seems frivolous. Where all programmes connect, the distinction seems unimportant.

SECTION VII. CONCLUSION

While the interprogramming approach suggests that programming languages have openings, as natural languages have "quotes", it is beyond this context to specify what they should be. However, their nature cannot be without implication, and the subject deserves particular design. Context openings govern the amount and form of access through a context, and are of linguistic importance not only within, but between programming languages. This is an important intention of the interprogramming approach. Not only would grafting rigid standard fittings onto languages be linguistically disharmonious, but for that reason survival of such fittings would anyway be in doubt. In the example contexts discussed, accommodations and redecorations suggested for interprogramming were just suggestions and subject to linguistic debate. Even so, several issues arose that seem likely to be of general concern in interprogramming context opening.

The structures of connection discussed were the common send-receive and request-reply models, but in fact many others and variants are possible. Even familiar applications of the discussed models are frequently more complex than might first appear. Some input/output arrangements seem strictly send-receive, for example, but do have provisions like the Fortran BACKSPACE or REWIND that complicate the coordination model. Some subprogramme arrangements seem strictly request-reply, for example, but allow communications via "global" data and so leave the bracket. And it does not seem necessarily true that familiar applications and structures need dictate the most appropriate approach to interprogramming. In conventional programming, for example, the "coroutine" structure, for all its seniority is seldom used explicitly (Knuth, 1968, credits Conway, 1963, but gives prehistory back to 1954). In interprogramming, the paradigm of switching from context to context might be more appealing. As outlined earlier, it is difficult to decide about structural superiority where contending structures can model one another. Linguistically, structural design need not really be that general: if a strict complex structure seems of sufficient linguistic importance it can always still be decomposed beneath the language in implementation.

There is further the problem of integrating existing context openings into a general interprogramming strategy. This is a question first of what should be considered openings. Is a time or date facility, for instance, beyond the context of an otherwise unconcerned language context or not? This can really be answered only in the language context itself. Probably most important in deciding about opening existing structures is whether the flexibility of interprogramming is useful or desirable there. So if time and date need never be influenced by some other context, there should be no need to open them. Even "files" never available beyond the one context would need not involve opening. Where an opening is desirable, there is then the question of other openings within the same context, and to what extent they should be integrated. While between languages this is the flexibility needed to make interprogramming general, within any particular context the distinction could be seen a linguistic concern. The send-receive model with "files" might be seen suitable for input/output, but a request-reply model might seem to better suit entry/exit. So they could be regarded as separate and disjoint openings, or they could be integrated. Integrated, for example, broadening the more restricted conception of "file" to a less explicit of, say, "streams" with no exterior form suggested. But while between programming languages no distinction can be made, within any one, linguistic sovereignty might insist.

Input/output, however, is certainly the most useful and common context opening generally available. Standardisation is important for easy interprogramming, and input/output does approach standardisation both in coordination and communication possibilities. In coordination, the "file" orientation has arisen because of the practice of storing and retrieving data from some device, so yielding a send-receive model but for such exceptions as device control. In communication, the importance of human interaction with programmes has lead to the inclusion of facilities for representing data in formats that model precedents in human application, yielding at least characters, usually lines, sometimes pages, and even books in Algol 68. These conventions mean that the easiest directly possible interprogramming will be through input/output or related context openings along those models. However, because these conventions do not follow programming precedents such as call-return, change or addition to the input/output context openings seems reasonable. The illustrations given for Fortran (READ/WRITE with EXTRINSICs) and Prolog (srequest-sreply) suggest one path for evolution, using a call-return coordination model but still making use of input/output facilities for communication.

A concern especially of interest with regard to the common input/output opening is that of communications granularity. The problem that seems frequent is simply knowing the exact granularity of some particular input or output. Where the input source or output target are simple storage media, the importance of the issue is small, but it is larger where they might involve human interaction, and is great where they might be pivots for interprogramming control. It is difficult to communicate further without knowing whether earlier transmission has yet occurred or not. Knowledge of granularity can be assured in several ways, most commonly by some understood quanta, or by explicit control facilities. Declaration of some constant quantum - "record" or "character" or some number thereof - does eliminate the uncertainty implicit in the possibility of unknown buffering, but does also require exact programme knowledge about the quantum, possibly difficult where data translation might have been involved. An explicit facility for granularity control is sometimes a simple "flush" to effectively conclude one transmission and commence another - though it is seldom available on input. The two approaches are related, and even seem so regarding the popular input/output paradigm: "file" can be a quantum itself, or "open" and "close" explicit parentheses.

The difficulties in finer control are surmountable, though if flexible interprogramming through input/output seems desirable, some language commitment seems necessary. Indeterminacy is undoubtedly useful in programming language definition, requiring no more precision than necessary in order to free implementation, for example, in differing implementation environments. At the point of flexible connection in interprogramming, however, indeterminacy cannot but be a hindrance. And as with input/output granularity, so with the "micro-parallelism" often seen in definition of the evaluation order of subprogramme arguments or expression terms.

Also often a concern in input/output is the provision of data translation or mapping facilities. Where such facilities are provided, they are often, as in Pascal say, tied to input/output itself. But the facilities are also useful in other interprogramming, and it seems unfortunate that if the input/output is not used, the facilities are less than immediately available. Where data translation is on offer, consideration of making it generally available seems worthwhile.

In programming language design for interprogramming, there seem many possibilities available for connection structure, communication granularity assurance, and the form of openings in context. Specific design of the form, the extent, and the integration of interprogramming, are linguistic concerns. As of course is any decision about interprogramming. But where such decisions and design proceed, there is a need for precision in language definition in order to enable and effect full and flexible advantage. There is a need to be unequivocal about the manner of equivocation.

In this new extension design there is one special danger. While interprogramming extension is unlike most other extension in that compatibility is the goal, there is little sense in diversity without intent. Regardless of the guarantees of interprogramming, languages relying on those guarantees must still all be established and supported separately in implementation. And while it may be reasonable in interprogramming design to make use of the way in which common support environments support the interprogramming, such design might be at cost in portability which could limit distribution and so availability.

For many common programming languages, interprogramming extension may be a refinement or simple use of some existing capability, particularly input/output or subprogramme call-return. Other programming contexts may require more indirect interprogramme handling or more specific interprogramming design. In the short term view, simple extension is probably the best approach for both existing and advent languages. Such experience may, in the longer term, lead to refinement in extension technique, an increase in reliance on interprogramming advantages in extra-language access, and so then on further programming language design itself.

CHAPTER VII
COLLABORATION: INTERPROGRAMMING OUTSIDE PROGRAMMING LANGUAGES

Collaborating between programming languages requires some provision within the languages themselves, but it must ultimately rely on a capability outside or between them. The environments exterior to programming languages have considerable variety too, however, and the generality and flexibility most desirable for interprogramming is less than universal. Some generality and flexibility is often available, and may often afford some limited or facility. As within programming languages, the next step is to examine some examples of these software environments, and explore how well suited they are for interprogramming or extension.

Outside programming languages are several software and hardware structures. Of greatest immediate concern is the implementation base itself, usually structured as a software operating system upon an underlying hardware (or at least non-software) machine architecture. In practical programming, the environment most commonly immediately exterior to any programming language is the operating system. In practically providing the interprogramming capability, and any nuance, this is the level most critically concerned.

It must be acknowledged that practical programming does not demand an operating system - does not, that is to say, demand any distinction between a programming language and an operating system. Smalltalk, for instance, is intended to encompass both. In his discussion of the design principles behind Smalltalk, Ingalls (1981) states: "An operating system is a collection of things that don't fit into a language. There shouldn't be one." However, it is not clear what the implications of this claim are. If the claim implies that any other programming language must be be implemented within such a system, there is no need dispute. It is an operating system, whatever else it is too. If the claim implies is that no other programming language is necessary or sufficient, then it's difficult to find a context for discussion. Interprogramming could be seen as an argument for having an operating system. Or at least for sharing between programming languages some schema of facilities traditionally provided operating systems. If a programming language does encompasses a sufficient network interface, that too could suffice. But at that point the taxonomy seems rather strained: for such an organisation to be workable in practice, the network structure might then resemble an operating system anyway. It doesn't seem very significant where any boundaries are between physical machines.

Keeping to immediate concern with conventional operating systems, the first capability required is concurrency of programming language contexts, and then facilitation of their coordination. In conventional operating systems or with any network involvement, communication facilities must be sufficiently flexible. If these capabilities are not sufficiently provided, extension might then be considered. And as with existing programming language design, so with existing operating system design must some importance be attached to provision of the capabilities in harmonious extension.

In exploration of these concerns, two examples of practical operating systems are here examined, Primos and Unix. These two systems are closely related in area of application and also in structural design. However, Primos is interesting in that it is not particularly well suited to interprogramming, though does offer mechanisms that might be used together to build some facilities. Unix is interesting in that it does just offer reasonable facilities for interprogramming, and moreover these facilities are important in the overall design - though the facilities directly on offer are still less than might be desired. Primos and Unix are interesting to consider because while they might first appear quite similar, they are in fact different in internal structural design - and importantly so.

In considering interprogramming between programming languages, however, the support levels beneath are not the only concern. Still outside programming languages, yet above the operating system level, are other programming languages. These too are worth some immediate consideration if allowing particular access to the operating system level, or if particularly tailored to easing interprogramming between other languages. After discussing the lower levels outside programming languages, attention is then also given to these "languages between languages". Some new such suggestions are introduced for combining languages together, and for assisting in data translation between languages.

SECTION I. PRIMOS

Primos is a general purpose timesharing operating system, designed and oriented about a now quite traditional view of timesharing: multiple user access via typewriter terminals, a general purpose interactive "command" interface, and a hierarchical file system. It is proprietary and commercial and is designed for the Prime range of computers only. The more general documentation is that by Seybold (1985) for the operating and software systems, and that by Hammond and Landy (1985) for the assumed underlying architecture.

Primos Structural Overview

Primos is a centrally managed system, and operationally consists primarily of processes, files, and devices - all under that central management. The process structure is provision for use of a terminal. It spans "login" for user validation, matching "logout", user accounted computing between those bounds, and some services beyond them; all these are directed through an interactive command facility. A more general image of this command facility is also available in the control language CPL (Command Programming Language). This is more part of the operating system than the process, and so is largely beyond process interference. Any individual process occupies a separate address space, hence neither is there any interference between processes. Process access to operating facilities is interactively via the command interface and programmably via a set of operating system implementing subroutines, ultimately via a "gateway" system call facility. Through the operating system, a process has controlled access to the elements under management: most significantly devices, files, and other processes.

Primos Interprogramming Opportunities

In considering interprogramming necessities, the first consideration is provision of some concurrency coordination, and secure coordination, between programmes. At first glance one might think that there simply is no facility in a Primos process for any programme multiplicity: there is only one address space per process, so per terminal, so per user session. However, Primos does provide a multiplicity of "command levels", for each programme, and the use of segmentation in memory structure does reduce - though not eliminate - the possibility of interference. Unfortunately, in earlier versions of Primos (before Revision 19.4 of 1985), there was a need for programmes to statically require specific memory requirements when components were loaded together. While some compatibility could be arranged in loading programmes specifically to cooperate, such control was not always possible and inevitably destructive conflicts arose. These important deficiencies are detailed in the manual by Burley et al. (1985).

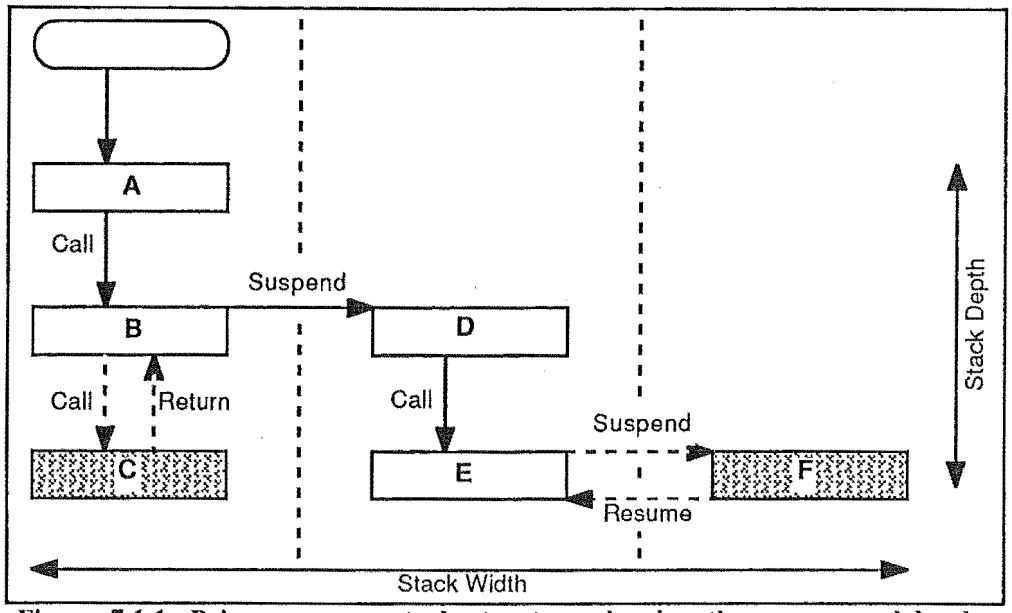

Figure 7.1.1: Primos process stack structure showing three command levels; programme call-return making stack depth, and programme suspend-resume making stack width.

Interprogramming Coordination Inside Processes

More recently (in Revision 19.4 and beyond) this particular problem has been resolved through restructuring to allow allocation of memory segments dynamically. This allows the previously difficult ability for a programme to "call" another and later "return" safely, accomplishing the hierarchical structure of programme coordination, all within any one command level. Unfortunately, however, the command levels themselves are strictly organised in a stack discipline, and each new level begins with the interactive command facility. So while it is possible to suspend a programme and later continue, it is not possible to suspend a first programme, begin a second, suspend the second, and continue the first. So there is still no process facility for the connection and coordination required in interprogramming.

If extension were to be considered, the programme structure could be made more general to allow arbitrary switching. The current capability seems to have been designed with human interactivity in mind, where a user might interrupt a programme, or a programme might suspend itself for hardware exceptions or programme reasons. Various explorations might then be carried out, and the programme later continued. The explorations might involve other programmes, hence the recursive structure of the stack. But it seems much more useful to be able to operate in a more arbitrary manner moving back and forth between programmes. And interprogramming connection would be possible if such moving back and forth could be programme controlled. Of course, there still wouldn't be great protection against interference between programmes, but that is the situation now anyway. If programmes reverted upon suspension to the "calling" programme, if programmes once "called" could be identified individually and continued individually, if the command facility were itself a programme of some sort, then the idea of command levels could be done away with. Old capabilities would remain while the desired new capabilities would arise. In practice it seems unlikely that the command level stack has ever been used very much anyway. In earlier versions this is understandable as programmes lower in the stack were most often corrupted by programmes higher. Even in the current

version, few programmes - including those provided as system support - include the necessary "on-unit" preparation to enable controlled continuance after suspension.

Interprogramming Coordination Outside Processes

If explicit extension of the system could not be considered, the alternative for interprogramming coordination is to step a level higher in the structure and represent "programmes" as entire processes. This is possible through the ability to dispatch from one process one or more "phantom" processes. Like all processes these model a "terminal session", but have no terminal (hence the name) and inherit user accountability from the dispatching process rather than by explicit login. This approach has the advantage that Primos has as much commitment in context separation as in terminal user separation: a great deal. An immediate disadvantage, however, is each process has all the services and environment of a terminal session - overhead for overkill. Operating system structures, for example, may not cope well with huge increases in the number of terminal sessions, phantom or otherwise. However, it secures as much latitude for "programming language context" as could be wished.

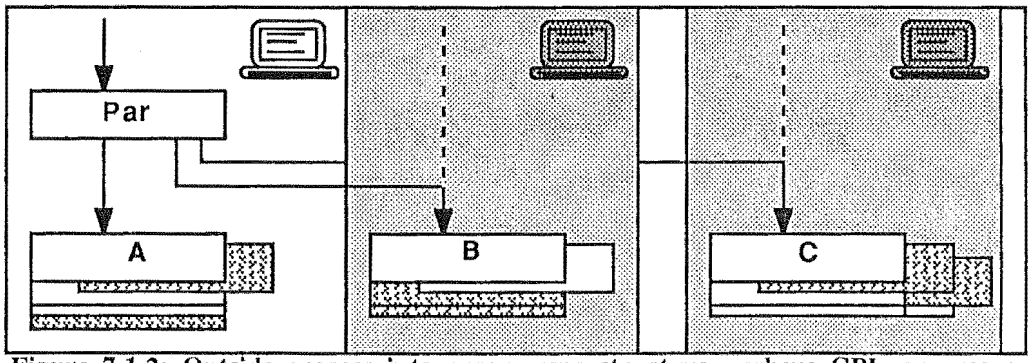

Figure 7.1.2: Outside-process interprogramme structure - where CPL programe "Par" dispatches "phantoms" for programmes B and C, then calls programme A directly in current process.

Unfortunately perhaps too much latitude: the terminal orientation and the great separation together cause further problems, as device and open file access are terminal session oriented. Accordingly, sharing open devices, even the terminal itself, would be a continuing difficulty in interprogramming. All that can be done without extension is restriction of individual device access to individual programmes, perhaps with such centralisation hidden from direct view by an intermediate software level of subroutine, say, between programmes and the particular device access routines. If some small explicit extension to the system is a possibility, perhaps device, and especially terminal input/output, could be made more flexible, but the orientation of the phantom process becomes becomes less analogous to a terminal session process, and a new structure altogether is probably indicated.

Using the approach in practice requires that several programmes be started in parallel phantom processes. The facilities to do this are easily available in CPL, and it is easy to write a small programme to accept programme names and arguments and arrange phantoms to swallow each. Normal Primos command line conventions can easily be duplicated in programme specifications processed in CPL because the

command facility and CPL are essentially the same. One programme can be distinguished to actually use the current process: this too is easily done from CPL, and that programme will have access to the terminal. When programmes within phantoms exit, the phantoms "log out", and notification is sent to the dispatching process. This can be detected, with sufficient preparation, by the CPL programmes, so it thus can continue until all parallel threads conclude.

Interprogramming Communication

If agreeable programme coordination could be arranged, either by extension inside processes, or by toleration or extension outside processes, the next consideration for interprogramming would be communication. The simplest form of communication is totally direct through primary memory. In the inside process approach there is neither memory protection nor contention between the programmes, and hence direct transfers might be used. To free programmes from the need and risk in directly negotiating, the direct communication might involve a communication segment accounted for in any bookkeeping separately to the programmes themselves, perhaps associated with another programme specifically used for coordination. The coordination required to support this is essentially coroutine or subroutine in nature: there is no other parallelism or pseudo-parallelism involved.

In the outside process approach the address space separation does not allow quite such a direct method. The operating system is common to any process, however, and Primos does allow for some address space shared by processes. However, such memory cannot be restricted to particular processes or particular users. This is rather a liability, leaving communications open to interference not only from accidental fumbling - as with the inside process approach - but also open to fumbling or malice from otherwise uninvolved processes of other users. Coordination of communication exchange in this arrangement would involve pseudo-parallelism, but Primos does provides primitive semaphore facilities that should be sufficient to ensure integrity.

Less direct but more secure communications would use files. The file system provides not only security at a level that already needs to be high, but also with great flexibility in access control. Both inside and outside process approaches could easily communicate via files, though the pseudo-parallelism of the latter requires more explicit coordination. However, not only is there some rudimentary coordination of file access anyway, but also the Primos semaphore facility is well integrated with the file system for specifying more particular coordination.

File input output is "word" oriented, as is all Primos and the Prime computer internally, on a 16 bit and two character word. Communication of arbitrary granularity would be no trouble, though, as the entire structure would need be especially provided for anyway. Any transparencies with interprogramming, such as with the genuine file input/output, would have to be handled at a higher joint level of abstraction. A serious efficiency disadvantage seems at first inevitable with a file method, and even the resulting permanence is unnecessary in interprogramming. However, like many operating systems, Primos maintains a pool of file associative buffers in primary memory as a file system cache. When reading follows shortly after writing,

file data transfer would be quite efficient. It should also be noted that Primos also uses paging, so any distinction between primary and secondary memory is more blurry still. With otherwise fast interprogramming therefore, the file system need not involve secondary memory speeds and inefficiency at all. And should great capacity ever be needed for communication, the facility is there.

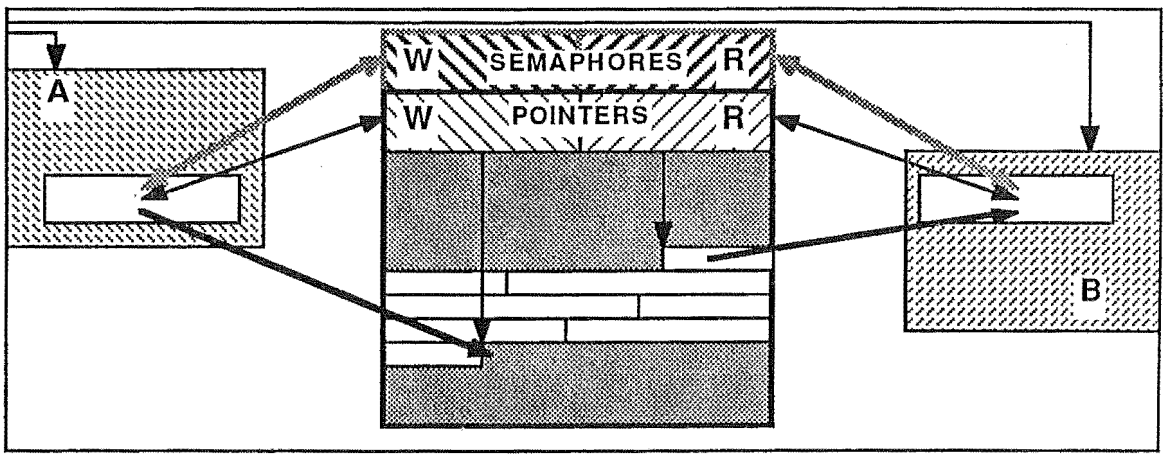

Figure 7.1.3: Primos interprogramming coordination and communication from one process to another through a file -the file can be used as a circular buffer of messages, with read and write pointers protected by file associated semaphores. Operations could also be overloaded on system subroutines used for normal file input/output.

In the matter of interprogramming communication labels, and in label matching, again an intermediate level of support would seem best. A set of subroutines similar to the Primos input/output system calls could be offered modelling some suitably general communications topology, and using some specified label. Communications with files or with processes could in this way be made somewhat transparent to the programme. In Primos, however, input/output with devices is not done in the same way as with files, so transparency might be regarded inappropriate. Within such subroutines, the label would be used to identify the programme of interest, and the actual detail of communication carried out as above. To accomplish the matching of label and programme there might be a requirement for some earlier preparation of an index, as with the message facility, or of suitable "mailboxes" in primary or secondary memory. It may further be desirable to "overload" this software level on the actual system subroutines themselves, and so, hide within differences in parameters, the differences in file input/output, and interprogramming. With such extensions for coordination and communication as discussed, programmes would then not need to be constructed in any different way for interprogramming or not. Use and inclusion of sets of subroutines would be necessary of course, but no alteration should be necessary should applications arise that would not require any other programmes. In the actual invocation of programmes, however, attention would be needed to assure correct interprogramming connections.

A final possibility for communications between programmes as processes is the facility for sending messages between processes. The facility was designed for textual messages between people, however, and the message length is restricted, and the precise coordination is difficult. In directing such messages, process numbers would have to be used, and so some index would need to be maintained between communicating programmes. Primos also has a networking component, Primenet (Venne and Fulchino, 1984), that provides

local and wide area services via X.25 protocols. This too could theoretically be used for interprogramming communication, but high overhead makes the speed slow enough to be bothersome. Moreover, few facilities are provided for maintaining separation between different processes and users communicating among themselves on one system - use of port numbers must really be pre-arranged and would be cumbersome.

Conclusion

If the Primos command level structure were made more general, the interprogramming situation would be much more acceptable. Programmes would then have common direct access to devices, most importantly the interactive terminal, there would need be less bothersome communication coordination, and the structure would correspond nicely to that of ordinary Primos programme usage. Not multitasking, and no inter-programme protection, but programme multiplicity none the less.

Some limited level of interprogramming in Primos is just possible anyway. The outside process approach to coordination can be used, and communications can be done with semaphore controlled file buffers. The problem with device and open file access will have to be tolerated, however, and that impacts programming significantly. And while communications via files seems the most appropriate, files are otherwise accessible, so there is also that possibility of inadvertent corruption of the communications structures.

For an interprogramming capability offering the flexibility and generality really desirable, however, Primos is simply not appropriately structured. In depending so much on single programmes at single terminals, the design cannot easily be extended to encompass the multiplicity essential.

SECTION II. UNIX

Unix (Ritchie and Thompson, 1974) is a general purpose timesharing operating system, though it is characteristic - and possibly important in its renowned success - that its commitments to any particular view of computing are few. The usual view is again multiple user access via typewriter terminals, a general purpose interactive interface, and a hierarchical file system. While proprietary, it has been predominantly non-commercial in distribution and application until recently. It has been a very portable operating system, and is probably available for a wider variety of computer designs than any other operating system. There are many variants of Unix, both because of the rather loose application of proprietary control, and because of much academic interest. There has been much recent work towards a non-proprietary standard, most significantly now in the IEEE coordinated "Posix" effort (IEEE, 1986), itself based on the two most widespread versions of Unix: "System V" (ATT, 1986), and the "Berkeley Software Distribution" (McKusick, Karels, and Bloom, 1986). However, Posix is not yet complete, and no other standard is currently dominant. In this discussion, therefore, principle concern will be for that common to all popular variants, both in system kernel and higher level software.

Structural Overview of Unix

Unix is a centrally managed operating system, and consists primarily of a process system and a file system under that central management. The process is an execution environment independent in memory address space and in scheduling, and has access to Unix through a set of system subroutine calls. Processes are created by the "fork" system call duplicating the process with only a indication of parenthood or childhood to distinguish the two. This is almost always soon followed by the "exec" system call to begin a new programme; the programme code comes from a file, and open files are shared with the parent.

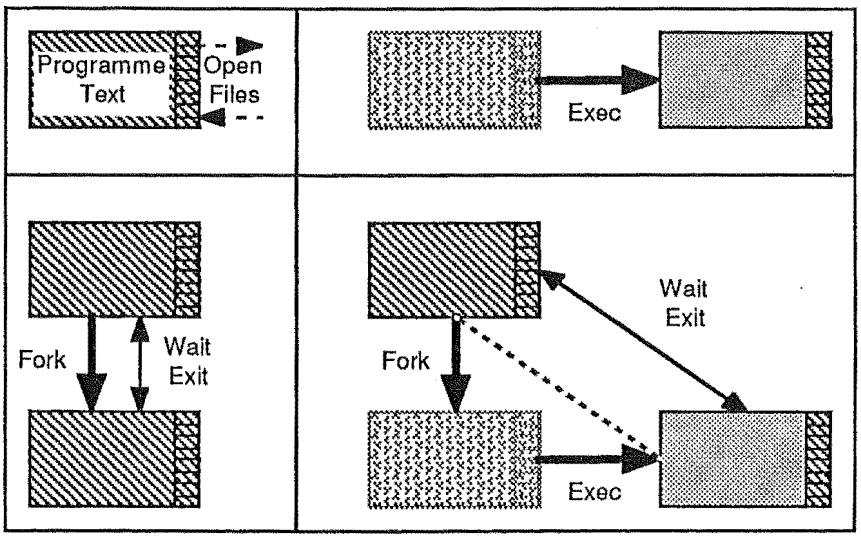

Figure 7.2.1: Unix Process Operations - "Exec" overwrites programme text of process, "Fork" makes duplicate of process. In both operations, open files are unchanged. "Parents" may "Wait" for "children" to "Exit".

Parent processes may use the "wait" system call to await the conclusion of child processes should they call "exit". In Unix a process is commonly used as a one programme envelope, and thus begins when a programme is to be entered, continues while a programme continues, and ends when a programme exits.

Processes themselves create processes and indeed the totality of this hierarchy is the totality of the operating system, typically as follows. The initial bootstrap programme load arranges data structures, creates one (other) new process, "init", and then itself schedules all other processes indefinitely. Init creates one process for each terminal, a "getty". A getty awaits the arrival of a user, "exec"s the "login" programme to verify access, and in accordance with user preference execs another programme - typically a "shell", the Unix command facility. In the shell, most commands are specifications of programmes to be executed. The shell arranges each programme execution as a new process. Any parent process may choose to await the conclusion of child processes before continuing. A command process typically concludes when the work of the programme is done. A shell process typically concludes when the user no longer requires interactive access. A login process typically concludes when the user no longer requires a terminal. The init process concludes when terminal access is no longer to be provided. The scheduling process runs until the processor halts.

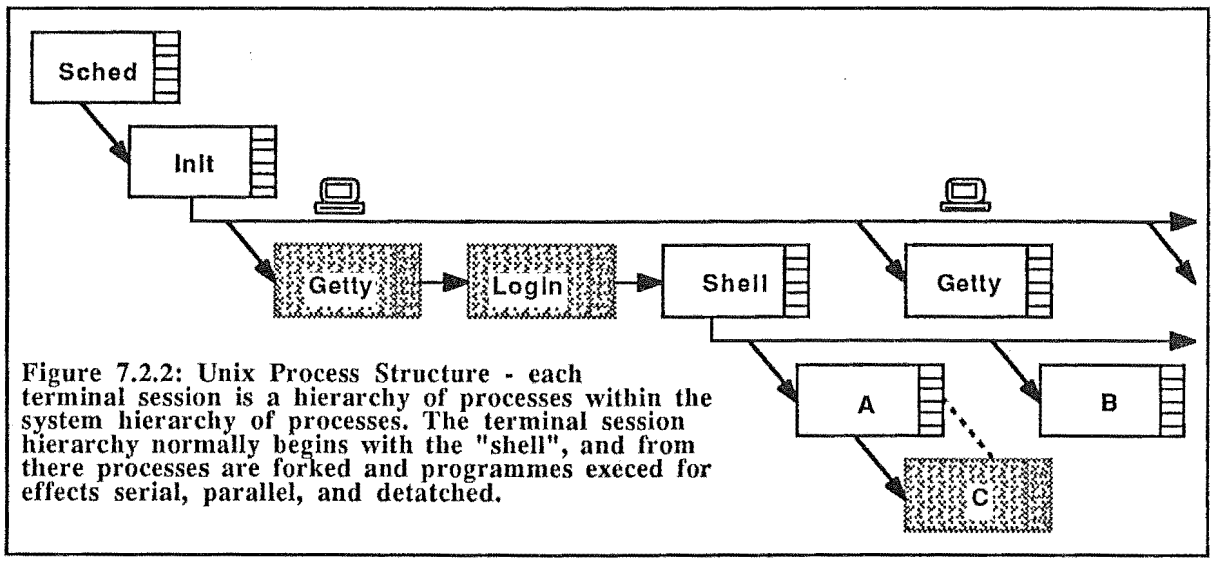

Figure 7.2.2: Unix Process Structure - each terminal session is a hierarchy of processes within the system hierarchy of processes. The terminal session hierarchy normally begins with the "shell", and from there processes are forked and programmes execed for effects serial, parallel, and detatched.

A process may attain access through system calls to the other operating system element, the file system. The Unix file system might better be described as an input/output system - the orientation is not about storing and retrieving but about directed data movement. The file system consists of "ordinary" files where storage and retrieval is the idea, "directories" where names are stored and retrieved imposing the hierarchical structure, and "special" files where the data traffic involves custom tailored system services, most importantly including input/output devices. All data is fundamentally "character" oriented, and all transfer is done through the system calls "read" and "write". Access to input/output concerns other than data transfer is provided through the "ioctl" system call, which offers services specific to a particular kind of file or device - changing terminal speeds, for example, or rewinding a tape.

Unix Interprogramming Opportunities

Unix clearly provides much that is of potential in enabling interprogramming. In looking first at coordination, the provision of a safe programme environment is straightforward: the process. More generally, as some languages might themselves use multiple processes, a programme might be represented by any such set - and some Unix variants provide for especially easy management of process "groups". The Unix process does provide the independence and protection useful, and does not provide bothersome overhead. The pseudo-parallelism offered is not strictly necessary, but the costs involved must be bearable as Unix already involves spinning off processes for each and every programme executed even serially. There are clear advantages just for that, as it affords protection between users, and between shell and programme, all in the same way. Moreover, because open files are shared transparently with and among inheriting processes, common access to devices from a group of processes is straightforward. Unrelated processes may even use the same devices still, but will be regarded separately. The high level of separation of processes does, however, mean that direct communication abilities will be limited. And the pseudo-parallelism means that the communication must itself be precisely coordinated.

Another problem involved concerns scheduling. Much in Unix is oriented about the process, and scheduling is "fair" between processes rather than between users - indeed there is little concept in Unix of "user" in an active sense. As a result, scheduling between users can be rather skew if process use is skew, and in suggesting greater use of processes this should be kept in mind. However, on the one hand interprogramming would not require pseudo "concurrency" of arbitrary interleaving, and on the other hand this is a problem Unix should face anyway. The "share-scheduling" user-oriented approach of Larmouth (1975, 1978) has been successfully applied to Unix (see Hume and Lauder, 1980), but is not widely used. More recently, Henry (1984) has developed solutions that work at the "set of processes" level with various controls; some AT&T Unix systems now feature this "fair share" approach to scheduling.

Looking then at communications between processes, there are several possibilities. At process entry some communication between parent and child is possible, as the latter begins life as a copy of the former. Moreover "exec" itself specifically does provide for data to be passed to the advent programme; this is commonly used as explicit ("argument") and implicit ("environment") data for programme entry. At exit too communication is possible, but very limited: one integer (the "status") may be passed through "exit" by the child and retrieved through "wait" by the parent. So an "entry" context opening is well provided for, as the hierarchical process-programme structure used by Unix all the time depends on it. But for other connections, and so for generalisation, other methods must be found.

Using the Unix "Pipe"

Unix does offer a direct connection between processes with the use of a "pipe". The pipe facility enables data to be output from one process and to be input to another, the data maintained in strict sequential byte order. Quite reasonable amounts of data (typically to 4096 bytes) may be transferred immediately, and a

blocking discipline allows more. Pipes are especially useful in that they are created by the "pipe" system call to appear as an open files, and are so accessed with the file system calls "read" and "write". This is very desirable and useful for any transparency in interprogramming, though that transparency does not extend to "ioctl", so programmes assuming some particular type of device may present difficulties.

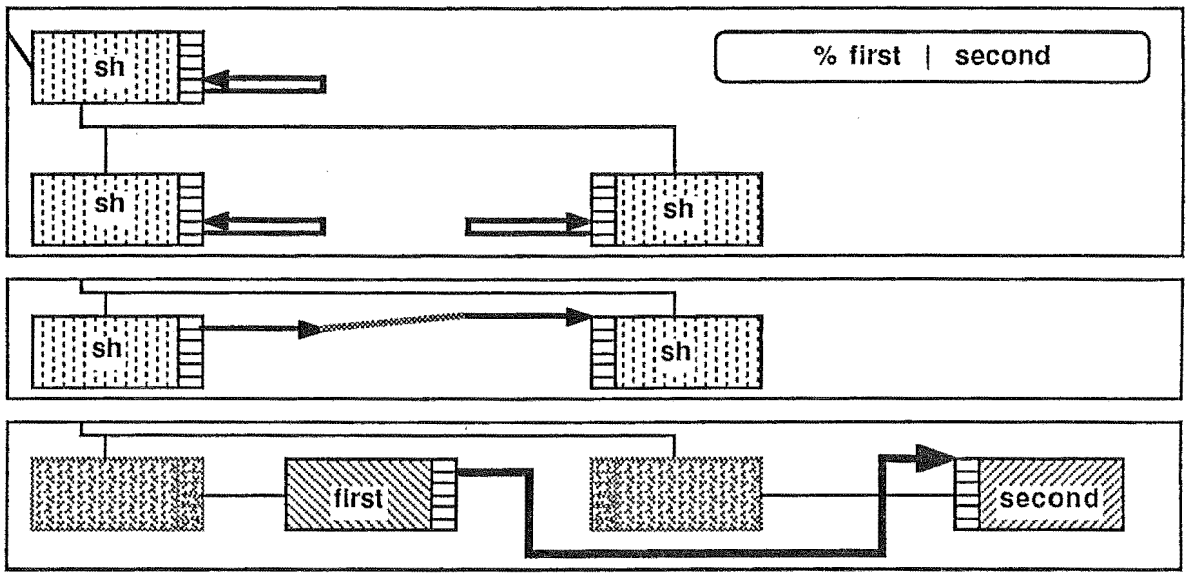

Figure 7.2.3: Unix Shell "Pipeline" between programmes - Shell first calls "pipe" itself, then forks processes for each programme, causing processes to share pipe. New processes each close one descriptor as appropriate, and "dup" other to standard descriptor. Processes then "exec" new programmes, which now have standard descriptors connected.

As an open file, a pipe is shared by a parent process with a child at fork. In this way, communication is possible between a parent and any descendent, or among descendents. Moreover, access is sufficiently similar to that with a file that in input/output, programmes may be unaware they are using a pipe at all. Higher levels of input/output facilities (such as the "standard" i/o library) can and in practice do apply structure above this, but this is easily changed where inappropriate. For example, block buffering might be applied where it is detected or decided that communication is only between programmes, and not between programmes and people. In programme pipelining this is usually right, but in interprogramming direct access to granularity control might need to be exclusive, so would need to be more programme dependent.

Pipes are used frequently in Unix, though almost entirely in their familiar role of coupling programmes from the shell. The role of the shell is important in practical interprogramming, and while it's reasonably clear that a user interface for interprogramming should present no special difficulty, some consideration is worthwhile. In Unix, the shell is an interactive command facility that is initially easy to learn, but also further encompasses detail sufficient for more complicated packaging of programmes. Interprogramming provisions might be offered as a separate context, as indeed perhaps the more complicated features of the shell should be. The direction taken in design so far is that of a single context, though, so no change appears immediately justified. Composite programmes relying on the shell to bind them, however,

will be less efficient than if bound by some more efficiently implemented, perhaps compiled, language - but this too is already the case.

The ordinary application of pipes in the shell is coupling of programmes in a very straightforward way. By convention, any Unix programme distinguishes one input file and one output file as "standard". When one programme alone is specified in a shell command, the programme becomes a process with standard input and output set to the input and output of the shell itself - interactively, for instance, some terminal device inherited ultimately from a "getty". When programmes are linked together in a "pipeline" shell structure, however, each programme becomes a process in parallel, and pipes are used to connect standard outputs to standard inputs in sequence as specified. So founded is the Unix idea of a programme as a "filter", one input means, processing, and one output means.

In implementation this is quite straightforward. The read and write system calls require an open file. These are specified by a "file descriptor" integer from zero to some limit. Descriptors are associated and disassociated with file system objects through the system calls "creat", "open", and "close". The "standard" input and output convention is adhered to simply by always reading standard input from descriptor zero and always writing standard output to descriptor one - not having explicitly associated either with anything else. The design is that "pipe" returns one input and one output descriptor newly allocated and functionally connected. Another system call "dup2" duplicates one descriptor from another, and is used between "fork" and "exec" to set child processes "standard" input and output. By arranging a sequence of programmes as a sequence of child processes with chained standard input and output, so the pipeline structure is created.

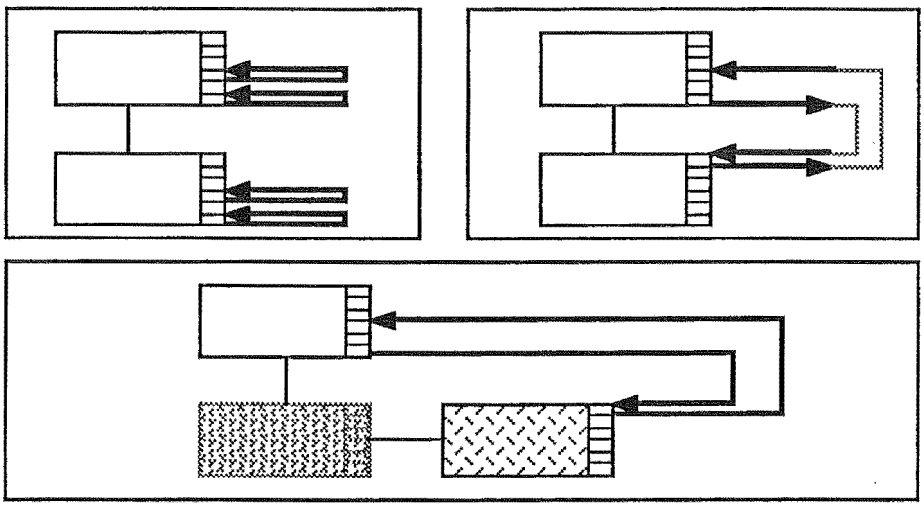

Figure 7.2.4: Using a Unix pipe to process serial data with another programme. The process first uses "pipe" twice, then forks. Input of one pipe and output of other are closed by the parent and the reverse by the child; child "dup"s descriptors to standard places. Child process then "exec"s other programme, which thus takes output and returns input to parent.

The conventional pipeline facility of the shell is a limited interprogramming facility itself, of course, but the "pipe" system call offers much more yet. At the programme level itself a variety of connection topolgies might be useful and explicitly created. Most common of the programme uses is in conjunction with

the facility to create new processes and programmes. Some Unix programmes, for instance, commonly creates a child process for a simple document formatter such as "fmt". Two pipes are created, the write descriptor of one and the read descriptor of the other are retained by the parent process, but the matching read descriptor and write descriptors are arranged as the standard input and output respectively of the child. The fmt programme simply formats what is read on standard input and writes it on standard output, so the main programme has a convenient formatting tool simply by writing unformatted text to one descriptor and reading it formatted from another. Some Unix variants have a system subroutine "pfork" conveniently packaging together the fork, pipe, and other components used in this method.

Extending Use of the Unix Pipe

In an ordinary pipeline, separate processes are able to communicate through pipes because all are descendents of the shell process that creates the pipes. But this need be no great restriction on the structure of communications. More general connections between programmes using pipes could be easily specified in an extended shell. With convention involving one input source and one output destination, however, only a linear structure like the pipeline can result. So to enable any new structure, some new conventions or rules will be necessary.

The first step beyond the strictly linear might be a recursive tree structure. Like a linear structure, a tree could be easily represented textually in the shell without a need for any "tagging" scheme, relying only on some nesting syntax. This could involve the syntax of names used to specify files in calls to "open" and "creat". By allowing a programme and arguments in place of a file name in the shell syntax, a recursive structure could result. Pipes could connect the standard input or output of a nested programme with some descriptor of the specified nesting programme. A similar but simpler service is provided by some interactive programmes already. Some editors, for example, allows "reading" of programmes distinguished from files by a leading "!": their standard output is read in. Some "mail" programmes permit mailing to programmes distinguished from addresses by a leading "|": the message is sent to the standard input of the programme.

Essentially a programme-masquerading-as-file must be dealt with in such a way as to ensure that a subsequent "open" system call results in an appropriately arranged descriptor being returned. One method might be to have "open" itself detect the masquerade and arrange for the programme-process, though this would make for difficulties should an opening programme wish to close and later re-open. Alternatively, the shell could detect the structure, establish the necessary pipe, and textually substitute the programme specification with something that a subsequent opening could appropriately understand. For example, "open" could recognise some syntax that defers to some particular descriptor and merely return that. The problem with this strategy as outlined is that in relying on programmes masquerading as file names it also relies on those names being planned for as entry arguments. So whatever the text is, it must be planned for, parsed, accepted, and so on - and not interfered with - by subject programmes themselves. This is really rather messy: the implementation of interprogramming support should not really be any concern of communicating programmes.

The use of standard input and output in implementation depends on the inheritance of descriptors and on the conventions about specific descriptors. This is used for both the shell pipeline structure and also for the simple redirection of standard input "<" and standard output ">". One version of the shell, the "Bourne Shell" (Bourne, 1978), does extend this capability by allowing any descriptor specified by number to be redirected. However, even ignoring that the syntax restricts redirection to single digit numbered descriptors, the approach seems rather crude. Numbers are neither very mnemonic nor very documentary in this circumstance, the ordering essential to numbers not here being of any concern. However, were names associated with descriptors the approach would seem more attractive. In that way both parent and child processes could have a descriptive handle on a descriptor, and in interprogramming these could serve as labels to match to connect context openings. The shell could establish any necessary pipes and associate specified names with the resulting descriptors; programmes themselves could then use whatever descriptors are associated with given (or derived) interprogramming labels.

So if the shell were extended to offer redirection in association with descriptor names, names could be provided for at each redirection structure. A descriptor could be established according to the particular redirection, and this descriptor then associated with the indicated name. This sort of approach is hardly new, separation of the concerns of programme and operating system in file naming through an intermediate level has been common for many years and in many systems. For example, descriptor names would resemble the "ddname" of the ubiquitous OS/360 Job Control Language. The ordinary redirection of standard input and output could remain done as now, though perhaps names could be established for those too: "stdin" and "stdout", say. If direct access to the descriptor numbers - as with the Bourne shell - still seemed desirable, those too could be provided for in a similar syntax. Indeed if access to the numbers did seem desirable, the numeric form could be the seen as the key syntax, and the name form simply a way of assigning a value to a shell variable. If a number were indicated, that descriptor would be used; if a name were indicated, an unused descriptor would be used, and its number assigned to the shell variable of that name. It would be necessary to change the shell to recognise this arrangement, however, as the current syntax does not allow for even the expansion of a shell variable to stand in the place of the descriptor number.

In exploratory implementation the approach could simply be a convention between an extended shell and descendent programmes. Descriptor names could indeed be shell variables and passed to child processes on entry via the "environment" data. Inheriting processes would then have the data to make the mapping between descriptor names and numbers, which could be offered by simple subroutines. Using this approach, the provision of a recursive tree structure in programme connection would not need to be dependent on arguments directly of concern to any subject programme. As before, programme specifications could be expressed in some nesting way to the shell, but only as the subjects of redirection. Redirection of input to such a nest would connect the standard output of the nest to the nesting descriptor indicated - by name, number, or default - and similarly redirection of output to a nest's standard input. This recursive structure also offers a different representation for the ordinary linear pipeline. So for applications needing only the simpler form a choice of expression there then might be offered a choice between the strictly left-to-right representation of a pipeline, and the more flexible tree structure offering prefix, postfix, or infix representation - or indeed any variety thereof.

It should, however, be remembered that a programming context may not use such a simple labelling strategy as so far assumed. A more complex approach would still be able to use pipes as the communication device, but descriptor naming facilities offered by the shell cannot be sufficient. However, the shell need not do all the connection work itself. Other programmes could be created to interpret whatever more complex connection description is required, and communication with those interpreting programmes could be carried out through the descriptor based connection approach. Beyond this, of course, other shells might be designed to cope better and explicitly with particular situations, and then interprogramming between such shells could be examined.

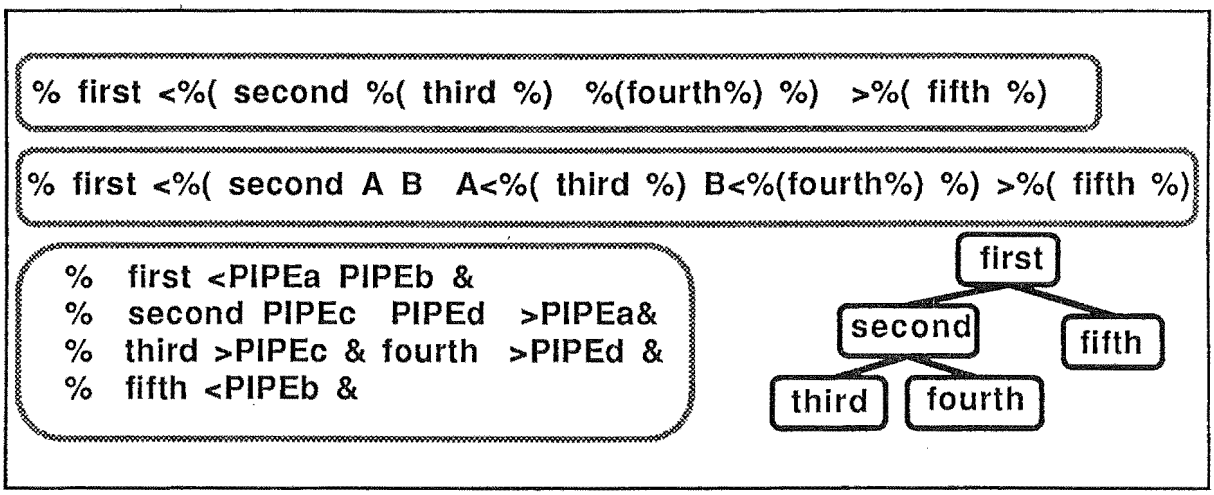

Figure 7.2.5: Representations of a hierarchy of communicating processes.
•The top line would depend on the "second" programme and shell correctly scanning programmes in %(%) syntax as files when necessary, and i/o subroutines arranging pipes and processes. Its shell would also have to scan %(%) notation as arguments when that was necessary.
•The middle line would rely on the "second" programme using i/o subroutines distinguishing named descriptors. Its shell would have to implement %(%) syntax and descriptor redirection.
• The bottom sequence of lines would depend on the "second" programme and shell using subroutines distinguishing tagged pipes. There would be no other changes necessary to the shell, and the approach would also apply for arbitrary graph communication structures.

The next general step in extending the common shell for programme connection, however, would be beyond the hierarchical to an entirely arbitrary structure. At this step, some tagging method would need to be introduced to the shell syntax in order to harmoniously represent in text the arbitrary (though directed) graph of communications. The tagging could be done by the shell, tags again being similar to - or again perhaps the same as - shell variables. Tags could be distinguished through syntax but otherwise used in the same circumstances as files (and so as of programme nests). The shell could create any pipes as necessary and pass the appropriate descriptors to programmes as indicated. Programmes communicating through tagged pipes would not automatically be established in parallel through that syntax, but would have to be linked with the shell background directive "&" explicitly indicating the parallelism. It is not possible for the shell in one pass, and perhaps interactive situation, to know which programmes should be established in parallel as connecting tags may arise in later programme specifications. However, the shell could possibly establish all programmes in parallel, except perhaps those not involving any tags, until all outstanding connections between tags are resolved. Anyway, the shell could retain tagged pipe descriptors until

connections are resolved, or perhaps until the enclosing shell variable environment concludes. With such a general structure, however, an arrangement involving more than simply the shell is really indicated.

Simple Implementation of Extended Pipes

The easiest and most useful simple way to implement tagged pipes in the shell is by intercepting calls to the system subroutines. This was inappropriate in the earlier case where in place of file names were (possibly recursive) programme specifications. However, simple tags do not present the same problems involved with potential multiple opening. This method does not actually need to involve the shell at all really, but merely the low level subroutines it uses. The disadvantage is that this does not allow the shell itself to create pipes early enough to share between descendents. The alternative is to create the pipes explicitly earlier on in a programme that itself execs a shell and so passes the pipes through the shell to share among descendents. This means that use of such pipes must be premeditated. However, this is not such a great disadvantage where the shell is not being used as an interactive environment but as a language to tie other programmes together. It should also be remembered that totally transparent device access is also only possible with and among processes inheriting the open files - other processes using the same device will be regarded separately.

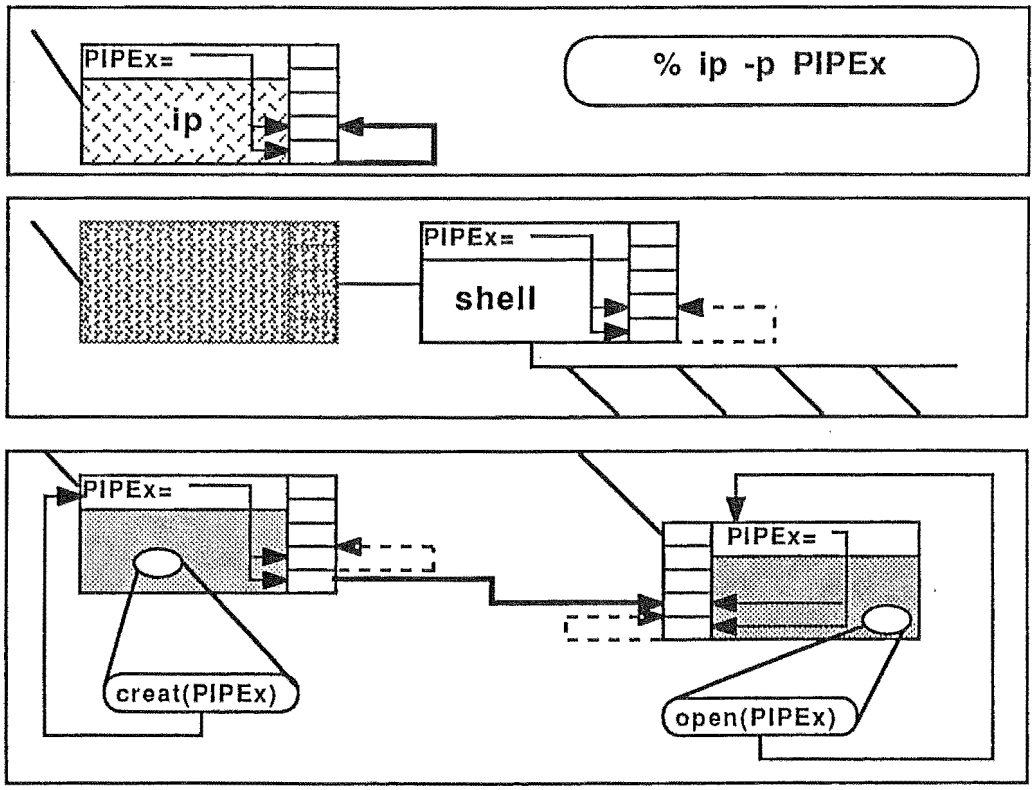

Figure 7.2.6: Simple Implementation of "tagged" pipes.
Setup programme "ip" looks at "-p" arguments, calls "pipe" as necessary, saves descriptors in environment variable named as "-p", and execs programme specified, the shell by default.
All descendents of "ip" inherit names and pipes, and "open" and "creat" intercepts return appropriate descriptors when pipe name is used.

The main advantage of the approach is that it not only avoids actually changing the shell, but for the same reason can be applied to any other programme. So by merely by re-loading existing programmes, and adjusting libraries used in programme compilation and interpretation, any Unix programme in any language has access to inter-process communication sufficient for some interprogramming.

The implementation has two components: the programme to set up the pipes, and the subroutines to access the pipes when needed. The programme to set up the possibility need only take a list of pipe tags, a programme to exec, and any arguments to pass to it. Once created, the pipes themselves will be passed on automatically as both "fork" and "exec" pass all open file descriptors. The pipe tags, however, will not accompany them directly, but can with the "environment" data be passed through programme entry at "exec". The environment is conventionally structured as a list of pairs of variable names and matching strings. In this case the variable name can be the tag itself, and the string can designate the file descriptors of the pipe matching that tag. In theory neither pipes nor tags are guaranteed safe passage though programmes as any programme can change its environment data and close or otherwise modify file descriptors. In practice, however, such interference is rare except where there is a clash with conventions for environment variable names or file descriptors. Both can be well avoided by adding rare elements to tags and not using file descriptors conventionally distinguished. Many Unix programmes depend on such minimal cooperation for successful operation anyway.

The actual subroutines of primary concern are "creat" and "open". These are the library routines that effect the system calls, so some care needs to be taking when replacing them that the system calls themselves are still available. However, this can be done by providing copies of the normal routines but with different names - these can then be called by the new intercept routines with the conventional names. The routines then need just first check in the environment to see whether the name passed is a tag, and if so return the matching file descriptor for the mode of access, reading or writing. If the passed name is not a tag, the normal system call can carry on as usual. This method is easily implemented, and is effective in all cases but those rare ones where programmes manipulate their environment unconventionally.

One of the very rare programmes that do not pass file descriptors transparently is unfortunately Joy's (1983) "C" shell, and in some Unix systems this is the most heavily used shell - albeit mostly in interactive use. The C shell does a lot file descriptor management to support it's foreground and background "job control" facilities, and it has no qualms in closing file descriptors it doesn't expect to find open. Fortunately a simple solution is effective: another system subroutine intercept, for "close", can protect tagged pipes for "open" and "creat" while satisfying the programme at large without further modification.

Another situation involving "close" has even greater potential for impact. Any process descending from the pipe set-up inherits the pipe descriptors, and potentially many of them are totally uninvolved in the intended communications. This does open a small door to potential interference, but it also means a difficulty in a process reading from the pipe in detecting the end of communications. This condition is only passed on through "read" when all the instances of the pipe are closed. And processes in general must keep the descriptors open in case a descendent process requires them. This means that while the method can be used

for communications between processes, transparency with ordinary file access would be less likely. A simple solution is again quite effective, however. The "close" intercept can write data to the pipe that a "read" intercept knowing the file descriptor to be a tagged pipe can interpret as a true end of communications. To a reading programme, this would be indistinguishable from the end of a file, so transparency can be retained. True transparency also requires intercepting "write" and implementing data "stuffing", otherwise trust must be put in a sufficiently rare close sequence. Moreover, because of the need to protect "close" itself against programmes like the C shell, the data sequence should only be written if in fact the descriptor has been used.

This is still quite a small amount of new software (see Biddle, 1987a), and the implementation of such an interprogramming facility merely by loading from a new library is very attractive. So attractive, indeed, that schemes involving more change, like the hierachical approach using named descriptors, seem generally unnecessary.

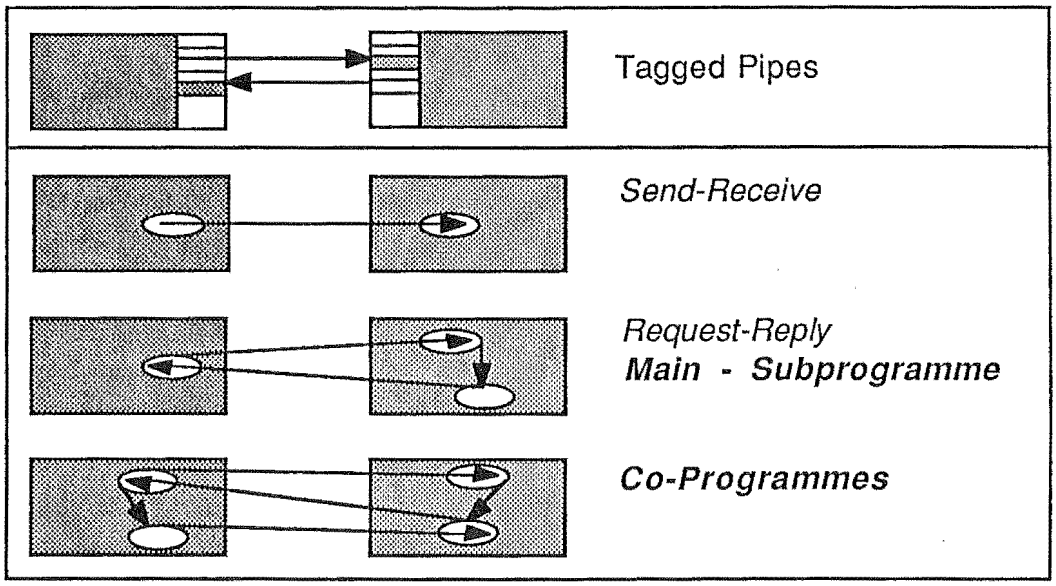

Figure 7.2.7: Using tagged pipes to build interprogramming connections - one pipe yielding a send-receive structure, two yielding a request-reply, and even more complex linkages. Use of input/output facilities means these are directly available for interprogramming, but higher level structures can also easily implemented using the same basic operations.

Input/Output Granularity

One of the concerns in interprogramming is the granularity of communication, a concern important in ascertaining ease in the communication itself and in coordination necessary to it, or resulting from it. The Unix pipe seems sufficiently useful for interprogramming that some examination be made of this next matter. The pipe facility follows the Unix file system generally in having very little data structure at all. The central unit of orientation is the "character", single size and appropriate to a letter or digit and canonically an eight bit byte. There is no accessible division below this, so any communication requiring this will have to make special provisions in using whole characters and specifying lengths internally. Above the character level there

is no division either, except to the end of communications implied upon "close". So in fact if communicating programmes are interested in any granularity other than one character or an entire pipe "full", Unix will not take responsibility for the structure. The philosophy here is that no one structure would suit all programmes anyway, so the one chosen is one useful in many applications and reasonably simple to implement.

Communication to a specific granularity is of course still possible with extra effort. It might involve communication first of a length in some understood format, then of sufficient characters. The "read" and "write" operations both work with the specific number of characters involved, so the case of "write" is straightforward. In the case of "read", this means a first read is necessary to discover the size, then a second - trusting the first - to retrieve the message. Such a method is easily encapsulated by the system subroutine intercepts described earlier. An alternative could involve explicit granularity marks inserted in the communications data. If reading is in units larger than a character, however, there is the danger of overruns, however, and of consequent inadvertent (at perhaps at subject programme level unforeseeable) deadlock. Reading in units of one character is unlikely to be very efficient. In many Unix variants, however, there is an easy answer in the provision of a "read" option that avoids blocking; some variants also allow data checking without actually committing to the "read" at all.

A further problem can affect granularity in writing, too. In order to reduce overhead, many programmes buffer output before actually calling "write". This is often inappropriate in interactive use, and the "standard" input/output library actually takes the care to attempt to determine if this is the case. Pipes are not regarded as a sign of interactive use, however, so use of pipes for closely controlled interprogramming requires some modification in the library. Transparent interception is again effective, and with "tagged" pipes determined from the "environment", ordinary pipes can still receive the more efficient bulk blocking.

In a Unix retrospective, Ritchie (1978) argues that input/output granularity structure of Unix is an important design feature, concluding: "In sum, we would consider it a grave imposition to require our users or ourselves, when mentioning a file, to specify the form in which it is stored". Clearly there is concern about fixed length records in older programming environments, but Ritchie's claim that "The notion of "record seems to be an obsolete remnant of the days of the 80-column card" appears too strong. There are differences between fixed and varying length records anyway, and for applications not so oriented, even the central nature of the (fixed length) character in Unix might seem inappropriate. Even within a character framework, while the Unix approach does seem to impose minimal structure, it also offers minimum assistance. Ritchie argues that this minimality is important to the mobility of character text files about Unix software. But while the mobility does exist, it is because many programmes (and programmers) so wish. Otherwise there is ample opportunity for them to provide incompatible structure additionally. And anyway, different layers of software could easily make different layers of structure available. Certainly is often a boon to quick and simple maintenance that so much in Unix is kept in clear text - the user information file "/etc/passwd", for instance. However, many Unix variants now keep more specialised structures in parallel to speed programme access; and maintenance of such parallel structure can be a nuisance. While it is true that Unix programmes do benefit from the common practice of keeping data as "text", this is only a convention, and programmes from other cultures often need to adjust. As outlined by Buckley (1987), there are various

practices involving tabs and line breaks in programming, and the Unix view is neither "standard" nor clearly superior.

Fundamentally, if programmes care about granularity, some additional structure, despite choice, is necessary. This is not a great difficulty, but it is an annoyance. Firstly, it is a slight break in the insulation between programmes: should one run amok and corrupt the communications structure other programmes may well follow and worsen the failure - though failure there would be, regardless. Secondly, it reduces the possibility for communications with Unix facilities without the need for translation - though a translation programme might be quite simple and re-usable. The ordinary Unix pipeline can be seen as a limited form of interprogramming: one limitation is the linear structure, another is the inability to coordinate through communications granularity. And of course the former to some extent obscures the latter. The problem can be worked around by information within the data itself, but for general interprogramming tit would be preferable to have some granularity control within the operating system seal.

Input/Output Generality

In proposing that programming language environments be explicitly opened with respect to that beyond, interprogramming does not need to insist that all openings be implemented in a uniform way. Connections between openings implemented differently can always be made with additional software, simply using tagged pipes with special interface programmes that dynamically connect to other programmes. And, as with data representation and programming language design generally, there is little point in increasing diversity without intent. Ideally, all connections of a programme with the exterior environment should be interchangeable in principle and nature, but this is a linguistic issue.

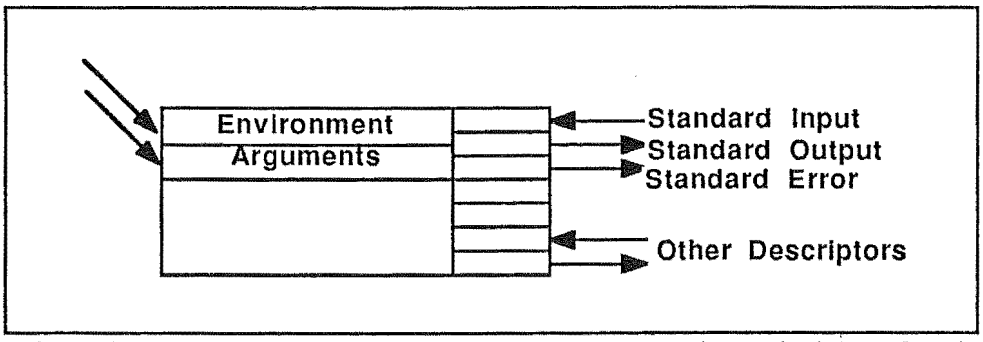

Figure 7.2.8: Unix process openings - One of the design principles of Unix is the independence of files, pipes, devices, and other special files: all are used through the general file descriptors. The shell `programme` syntax extends even this model by allowing redirection of one programme's standard output to another programme's entry arguments - a similar environment syntax could be possible too. It is tempting to consider integrating the "exit" return code, and perhaps even asynchronous signals as well. But the data structures are not always compatible or even applicaple. Translation might be possible, either by other processes, or in system routines like those in Ritchie's "streams", but is the generality necessary? The question is whether an operating system should be a linguistic structure, and able to offer the conceptual framework that enables.

In practice of course, as with any interprogramming recommendation, this need only be adhered to as long as the benefits of interprogramming are of concern. But it does mean that consideration should be given to the generality of connections to context openings of a Unix programme. The communication facility outlined using pipes is in fact sufficient for any context opening by itself. However, in normal Unix operation both communication with the file system and with the shell "command line" are so common that integration might be considered. And perhaps an exit opening might also be contrived to match the entry opening.

Communications from the shell to individual programmes could be offered in exactly the same way as with named descriptors or tagged pipes: the connections simply distinguished by prearranged names, but passing the same information as argument and environment, and retrieving exit status. In implementation, the common interface could simply be provided by the name or tag facility itself, though this might bring unwanted responsibility within the programme environment. More generally, the shell could actually offer and accept communications with programmes by pipe directly through descriptors, using a prearranged descriptor convention like that used for standard input and output. The existing entry argument opening could still be provided as well, so no capability need be lost.

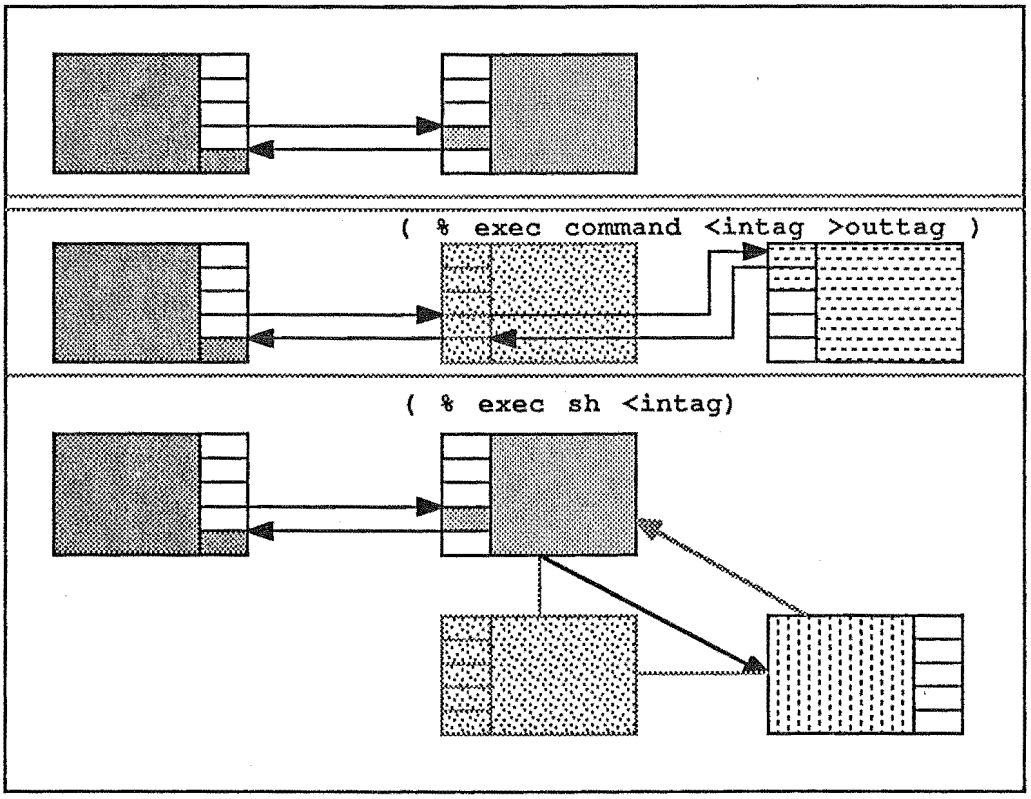

Figure 7.2.9: Generality of openings imitated via intermediate programme - A programme connected via pipes can dup the pipes to standard input and output, then execute another programme; it could also collect data from an input pipe, then fork-exec another programme with the data as arguments (it could even wait and return exit status via the output pipe). In these ways, programmes do not need any "shell escape" mechanism to be able to dynamically connect with other programmes. In practice, both can easily be implemented from the shell: in the first case simply issuing a command with input and output redirected to named pipes; in the second case simply by taking input from the tagged pipe itself.

In looking at integration with the file system, it can be seen that some facility would be available anyway. If named descriptors could be redirected, as numbered ones can, then the subject of the redirection could simply be a file, so allowing access to files through named descriptors. More directly, programme use of a named descriptor that was not in fact established could result in connection with a file of that name, so allowing some transparency between file names and descriptor names. It would need to be remembered, however, that internal file structures would not necessarily be arbitrarily interchangeable. In the case of tagged pipes, generality with ordinary files is always available by additional small programmes connecting pipes with files: the small concatenation utility "cat" would even be suitable.

These extensions discussed are really reasonably simple, and similar ideas have probably occurred to many Unix programmers. One strongly related idea is that of Shultis (1983), who proposes a "functional" shell that offers these sort of facilities. For example, it generalises the pipe idea to that of "labelled" and "structured" "data streams". Whereas the aim here has been simply to show how the necessary facilities could be built reasonably smoothly into existing shells, Shultis is proposing quite a different facility, more totally oriented about programmed rather than interactive use, and more mathematical than operational. It's similar, as he points out, to the proposal of Landin (1966) for Iswim, though whereas Iswim was suggested as a mathematical base language beneath other programming languages, the FShell would be above other programming languages linking them together. The particular proposal is notationally complex, and requires in full a special character set. Shultis argues for the benefits of such a step, but seems himself pessimistic about the implications. While that might once have seemed inevitable, however, the possibilities now available with economical fast display devices perhaps indicate such ideas might be pursued after all.

Unix Beyond Unix

All that has been discussed so far involves no change in the system call defining kernel, and applies to any Unix variant. In an extended Unix, other choices would probably be available. Even new methods for concurrency might be available. Berkeley Unix, for instance, allows processes to be stopped and restarted, so precise coordination might be built in, rather than using completely independent scheduling. However, new methods for communications are more common. For example, inter-process communication through ordinary files would be a possibility were some coordination facility on offer such as the "locks" suggested in the proposed Posix standard. At a more primitive - though efficient - level , some systems also offer shared segments of memory, and semaphore provisions to allow coordinated access.

In some Unix variants it is possible to include pipes in the file system name space. This allows a pipe to be created and opened by name from any two independent processes wishing to communicate - details of the semantics vary. Such "named pipes" are often a simple extension of the kernel because the ordinary pipe is often implemented as a file but with a circular data buffer, and not named in any directory. Were named pipes available with similar semantics to the tagged pipes discussed, the tagged pipes would be largely unnecessary. In a similar vein, some Unix systems offer "pseudo-terminals". These are pairs of special files, a master and a slave, where input and output of one are connected to output and input of the other. The slave is indistinguishable by a programme from a "real" terminal - all "ioctl" calls work. And

because another programme can be reading from and writing to the master, the pair together enable a programme to connect with another that assumes or insists that it will only deal with a terminal. This is accordingly a way to add an interprogramming capability to an existing programme that uses "ioctl" in ways incompatible with a pipe. There are still other devices with a similar problem, but this is the most common one. The pseudo terminals are typically just implemented as separate special files, however, and the overhead and need for processes to themselves find available pairs is cumbersome. Flexibility in connecting processes and "files" of varying kinds would simplify many possible solutions in networking and windowing, however, and Ritchie (1983) describes an approach based a dynamic capability to insert system processing modules between processes and files. This "stream" generalisation would address the "pseudo-terminal" and much besides, but is still largely experimental (see Bach's description, 1986).

Other more general and explicit message passing capabilities have also been proposed for Unix. Some real-time oriented variations like Mert (Lycklama and Bayer, 1978) provide specialised facilities, but facilities for integration with the usual system have also been introduced in simple straightforward approaches like that of Blair, Mariani, and Shepard (1983). Most general and wide ranging of capabilities, however, is that introduced in the Berkeley distribution 4.2BSD (Leffler, Joy, and Fabry, 1983). This approach involves many possible communication arrangements in one large structure for connecting processes with "sockets" encompassing scale from the ordinary Unix pipe to the Arpanet. The common technique is to use the input/output system calls in message passing, so preserving the "generic" aspect of Unix data flow (and certainly of assistance in generalised interprogramming). The BSD approach also offers a send-receive model for communications beneath the read-write level, however, and thereby allows total control of granularity. Message passing schemes generally also do allow communications between independent processes, so requires no special communication set up by a common ancestor process. However, the BSD method is not symmetrical, and some negotiation is necessary for processes to establish "server" and "client" roles. Moreover, while the BSD method does permit use of the "read" and "write" system calls, and while sockets are named, sockets are never formally in the file system name space. Accordingly, transparent treatment of files and sockets still requires some "tagging" modification. This can be easily done by intercepting "open" and "creat" calls as discussed before. No special preparation is required in this case, but the server-client determination must be made. In a general case, however, this can be done simply by attempting to be a client, and becoming the server if the attempt fails (see Biddle, 1987a).

Because of the generality of the whole system, it is also possible to directly send or receive data with exactly the mechanism to processes on different systems. This can be done within the same transparent subroutine layer, or can alternatively use separate small programmes that connect local input or output with distant processes. The messaging facilities in BSD also address the problem of transparent device access between processes without prior opening and sharing of the file descriptor. This is done quite directly by a processes being able to "send" and "receive" open file descriptors, and is a very powerful facility. In fact, however, the transparent access is really a side effect: the stated intention of the ability is the passing on of file access rights, and passing the open file descriptor was just a convenient method.

These Unix extensions do provide useful facilities for interprogramming. However, such generalisation, once begun, might proceed far and wide. With named pipes there would be no need for

tagged pipes. In fact, no need for ordinary or "temporary" pipes - there is no special provision for temporary files, after all. And if named descriptors are to be interchangeable with files, perhaps they might in fact be a new kind of "indirect" file. Files themselves, not just descriptors, would then be subject to redirection. And as for prearrangements like "stdin" and "stdout", they too might be special files - it's not all that different from the handling of devices like the special file "/dev/tty", general in name but specific in use to refer to any user's terminal. Special files "/dev/stdin" and "/dev/stdout" are in fact a common local addition to Unix systems, along with the equivalent files for every available file descriptor, to ease specification of some already open file to a programme insisting on a name.

This in turn is reminder of the generality already existing in the Unix file system: names of special files refer more to their implementing programmes, kernel resident though they be. Much of the flexibility in Unix stems from the overloading of the main operators on the "file" system, "read" and "write". This overloading, and the data typing that enables it, might be applied further still. Why, for example, should a directory need particular system calls beyond "read" and "write"? If they were overloaded especially to deal with directory structure while preserving integrity, it would not be necessary. And if some "special" programmes can be used through file access, why not any programme? In the recent Unix variant "Minix", Tanenbaum (1987) implements virtually all system subroutines with only "send" and "receive" system calls anyway. What is the difference between "files" and "programmes"?

It's difficult to tackle such questions definitively without again stepping into the very deep waters of linguistics. It is true that programming and operating structures might be made more and more general, as suggested, to a blur. But they need not be. Total object orientation might seem tempting, but the simplicity of a more restrictive structure might sometimes be more helpful. Anyway, objectification itself contains subjective elements: there is no universal definition of what constitutes an "object" after all. And it is not clear there can be total abstraction without total reduction. Unix can already reasonably easily support the generality, the immediacy, and the flexibility needed for interprogramming. While the hierarchical process structure sometimes seems an imposition on free structure, it does allows easy control of very common structure. Some extension might be useful, indeed total reorientation may be useful, but it is not strictly necessary. It is beyond the scope here to discuss whether an operating system should be anything but a network of processes.

Conclusion

Unix has been and continues to be very successful in use and in application: unprecedentedly so for its portability and initially very limited commercial support. There are many reasons involved in this success, and it is difficult to isolate the dominant factors. While claiming several technical reasons (1978), Ritchie also admits various sociological influences (1984). An important technical reason is the potential for what Kernighan (Kernighan and Pike, 1983) calls "programs to connect programs": the generality of the files, devices, and pipes. The control and coordination of connecting programme input and programme output with the Unix pipe does yield great flexibility both in interactive use and complex application design.

Yet this programme coordination approach is clearly not as flexible as interprogramming might be. And while the orientation about "standard input" and "standard output" asks little of programming language design, it is not without impact on programme design itself. Programme input or output might concern only a connected programme, but also may concern a connected human user: a single design often fails to satisfy both connections. This difficulty, and particularly the practice of satisfying programme over human user, may be the root of some user hesitance about the Unix design. Norman (1981) concludes: "the truth about Unix: the user interface is horrid". The real truth is: often the user of a Unix programme is another programme - and the user interface is then very good. Flexibility is the "problem", and in the software design, more - not less - flexibility is called for. The additional flexibility required is available if the input/output name space is used to connect not only programmes and files, but to connect programmes with each other. Limited interprogramming might be a key to the success of Unix; this more general interprogramming might further that success.

SECTION III. PROGRAMMING LANGUAGES BETWEEN PROGRAMMING LANGUAGES

While the principle concern of interprogramming outside directly involved programming languages must be for the support needed from operating systems, there are other concerns. There are programming languages outside of those involved in direct application programming of any kind. These languages do, however, involve application languages in several ways. There is no established taxonomy as this again largely an area of exploration. Three significant kinds of language are discussed here: "meta-programming" languages for programming with other programmes as components; "programming language multiplexors" for offering differing languages within single texts; and "inter-languages" for translating data communicated from one language context to another.

META-PROGRAMMING LANGUAGES

In an environment where programmes written in other languages can be accessed in a flexible manner, it becomes possible to create new languages based heavily on that access. The first such "meta-programming" languages were probably the general control languages discussed earlier, like Primos' CPL or the Unix Shell. Both these can actually be used as simple programming languages alone, but their significance is really in how they can connect other arbitrary programmes together into a new whole. Some such languages are tightly bound with an underlying operating system, as CPL is with Primos, and the capability is greatly limited to that one privileged language. Others, however, like the Unix Shell, are not privileged in any way; any programme or programming language has the same access capabilities. This latter circumstance makes it possible to have a variety of such languages available. As mentioned previously, this means that many Unix systems have two or more versions of "the" Shell. While the shells popular on Unix all share a common heritage, others have been proposed: Shultis' functional FShell has been mentioned, but there are also shells based on Lisp (Ellis and Levin, 1980), on Snobol-Icon pattern matching (Fraser and Hanson, 1983), and others. Some large integrated systems, like the "Emacs" type of extensible text editor (Stallman, 1986, for instance) include many services of the shell themselves. But a general programme connection facility also makes quite new and specialised meta-programming languages a useful possibility.

Perhaps best known of such specialised programming languages in the Unix environment is "Make" (Feldman, 1986). Make is an aid to maintaining large programmes, and contrives to "make" components derived from others when the others are modified. It's principle is that programmes are based on sets of files, and that some files are used by some programmes to create other files. It therefore depends on the modification times associated with files, a dependency hierarchy detailing the dependencies between files, and a script of programmes to be run in making files higher in the hierarchy from those lower down. Make therefore needs to be able to run these programmes itself, so "making" files up to date. In this way, Make is a meta-programming language depending on the Unix ability of easily dispatching new programmes (as processes) from within a programme. Moreover, should any programme under its control fail, Make itself is

fully insulated by the address space separation. It would be possible to write Shell programmes to do the same job as Make, but Make is more efficient than Shell programmes would be, and is much better suited for its application in a documentary way. It's very widely used in the Unix programming community. In Primos, CPL programmes are used for this purpose, and they are much longer than Make programmes, less efficient, less flexible, and much more difficult to write and understand. The flexibility and documentary situation is so bad, in fact, that it is probably worthwhile neglecting the efficiency argument and actually implementing a simple version of Make in CPL itself. This can be done quite effectively in about a hundred lines of CPL. But the efficiency and better syntax, diagnostics, and error recovery of a compiled meta-programming language would still be more desirable.

Such languages can be applied to various specialised situations, and can even extend beyond single systems. The small language "Rap" (Allan et al, 1987) was designed to flexibly connect one system with another for automatic login, file transfer, and other simple service interactions. Rap programmes on a Unix system use a "special" file implementing characters to and from another system, like the Unix networking facility UUCP (Nowitz and Lesk, 1979) does - in fact Rap was inspired by the UUCP login sequence. Rap was in Control over the remote system is purely through the character streams, but on the local system arbitrary programmes can be accessed. Rap itself directly provides only facilities to send character strings and distinguish character strings received. All other capabilities come from other programmes, which can be connected with the character streams to and from the remote system. Rap itself therefore provides the coordination necessary in matching systems without special protocols, and then the connection necessary in coupling local and remote programmes.

```
set timeout 60;

try 5 {   send "\r";
          expect { "name:": ; } }

send "ouruserid\r";
expect { "word:":  send "ourpassword\r"; }

expect { "$": {    send "@mailtofile\r";
                   expect { "$":      send "kermit\r"; }
                   expect { "32>":    send "send ours.mai\r"; }
                   shell "kermit  ••• -r";
                   expect { "32>":    send "receive\r"; }
                   shell "kermit  ••• -s theirs.mai";
                   expect { "32>":    send "quit\r"; }
                   expect { "$":      ;
                            default:try 5 {    send"quit\r";
                                               expect{"$": ; } } }
                   send "@filetomail.com\r"; }

try 5 {   send "quit\r";
          expect { "$": ; } }
try 5 {   send "logout\r";
          expect { "name:" ;} }
```

Figure 7.3.2: Rap programme for simple sutomatic file transfer for electronic mail - Another system accessible by serial line is logged into, an inward mail file created and transferred, an outward mail file transferred and dispatched, and the logging out done. The "send" directive sends data to the line, and the "expect" directive receives data from it. "Expect" can distinguish several possibilities and a "default" case on timeout. The "shell" directive executes programmes on the local system. Only primitive iteration is provided by the "try" directive, which repeats while "expects" are unsatisfied to some limit. Outside a "try", such unfulfilled expectations end the current scope. [Example slightly abridged from actual Unix-VMS mail transfer programme]

This is a quite straightforward facility that yields great flexibility in connecting computers systems without needing special preparations. Rap programmes have been used to connect systems together automatically to exchange electronic mail, to transfer files, and to run files of commands. Programmes may connect directly through a Rap programme, or may connect through other file transfer protocol programmes like Kermit (da Cruz, Tzoar, and Catchings - 1983) to increase reliability of the serial connection. The distant system need not be Unix or anything in particular, but the local system does need that essential facility of accessing arbitrary programmes and connecting them together.

A PROGRAMMING LANGUAGE MULTIPLEXOR

Even where diverse programming languages can indeed be used together, their textual separation and independence might be inappropriate. Where one programme accesses a programme in another language, for example, it might be mnemonic if the accessed programme could be textually inserted as if it were not a different language at all. This is a linguistic issue, and may or may not be appropriate in particular circumstances. Where appropriate, however, an eclectic linguistic harmony can result. Practically, it can also mean fewer individual programme source components, and possibly more easily understood and maintained programmes. Combining different languages textually - programming language multiplexing - can result in a mixture reminiscent of such "multiple" programming languages as PL/I. The textual combination, however, is merely an immediate visual juxtaposition, and so proof against the complexities of overcomposition.

Various pieces of software have had the ability to distinguish and separate different sequences of characters in single text. Some macro processors (Kernighan and Ritchie's M4, 1976, for example) allow macros to expand text out of input order. Some document formatting programmes (Ossanna's Nroff/Troff, 1976, for example) allow text to be diverted to buffers and later expanded in different contexts. Knuth's Web (1984), however, is an explicit multiplexing of programming language and document formatter. Web allows integration of documentation that can be formatted automatically, and of source programmes that can be automatically implemented. In this way, programmes intended for publication, for example, need not exist in parallel with copies for actual use - thereby avoiding dangers of parallel maintenance.

An attempt at a general purpose multiplexor is "Mux" (Biddle, 1987b). Mux allows for the division of a file into "sectors", and for the "selection" and ordering and extracting of such sectors into new files. Sectors are labelled, and are selected by label, and both sectors and selections may specify files to be produced. In this way, a Mux file contains the information to construct other files according to file extraction specifications. Sectors are distinguished by a "sector" directive. They are labelled explicitly, and continue textually until a closing null sector directive, or another sector directive - whereupon the sectors are nested. Files specified in the sector directive govern what is produced by the sector. If files are specified, the sector applies to those files only. If no files are specified, the sector applies to no particular file, but to any files involved in selection. Selections are distinguished by a "select" directive, specify a sector label, and imply the

production of that sector. If there are several sectors of the same name, they are produced in their order in the file.

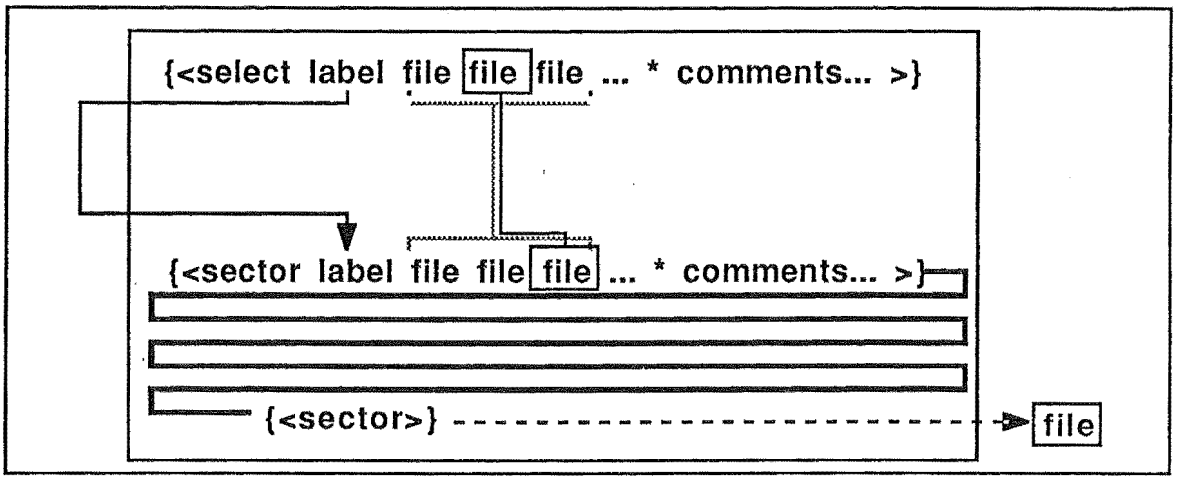

Figure 7.3.2: Structure of Mux directives - There are only two,"Select" and "Sector", and together they determine file extraction- hence demultiplexing the multiplexed file. Both directives are in the file itself, so the structure of ordering can be entirely determined by syntax mapped into directives. A "select" extracts matching named "sectors", and generates data for files mentioned in selection and also extraction. An absence of files mentioned implies deferral to the matching directive. "Select"s are active, so inclusion of one within an already selected sector implies recursive generation. "Sector"s are passive and nesting of a sector within one selected does not imply recursive generation. The programme argument line allows for specification of both external selection and sector directives, plus optional macro processing.

... num = ...	... num = ...
numline = right(num, 5) write(tofort, num) {< sector >}{< sector fortran >}	Fortnum(num, 5, 0, NUM)
READ(10, 100) NUM 100 FORMAT(F5.0) LOGNUM = ALOG(NUM)	LOGNUM = ALOG(NUM)
WRITE(11,101) LOGNUM 101 FORMAT(F10.5) {< sector >}{< sector icon >} lognumline = read(fromfort) lognum = trim(lognumline)	Iconum(LOGNUM, 10, 5, lognum)
... = lognum ...	... = lognum ...

Figure 7.3.3: Fragments of Mux files of interprogrammed Icon and Fortran - using simple input/output openings. On the left, Mux sectoring and data translation is explicit; On the right, M4 macros "Fortnum" and "Iconum" incorporate such details.

Files specified in the select directive govern what is produced by the sector. If files are specified, at most those files are produced from the sector, possibly limiting the files specified at the sector. If no files are specified, the selection applies to any files specified at the sector. In order that files may be produced with

careful control over format, Mux directives may begin anywhere on a line, span lines, and end anywhere on a line.

Mux is applied to multiplexed files with a command that does the necessary sector extraction. The command also allows further selection criteria, so only certain sectors need be extracted should that be desirable. The flexibility of Mux is much greater when a macro processor is also used to suit the syntax better to the languages involved. Accordingly, access to both M4 and the C preprocessor is built in, and macros may also be defined by optional parameters to the command.

Mux is useful whenever the order or presentation and intertwining of different components of a programme is important. A very simple example would be the re-ordering of a Pascal programme to place less important procedures and functions later in the programme. And because files may weave together in some sectors, Mux provides a general alternative to Web: should programme source code and documentation cohabit the same files, source could be extracted and compiled, and documentation extracted and formatted. More generally, Mux is useful whenever joining dissimilar notations, enabling points of logical meeting to be physically juxtaposed, so slipping between the different rules of the different notation systems.

```
{<select hello >}
{<sector hello hello.c >}
/*
{<sector hello hello.c hello.doc >
"C" programme to greet the world.
{<sector >}
*/
#include <stdio.h>
/*
{<sector hello hello.c hello.doc >
The function "main" is the main
funtion.
{<sector >}
*/
main()
{
/*
{<sector hello hello.c hello.doc >
Printf is the print function.
{<sector >}
*/
 printf("Hello, world!\n");
}
```

hello.doc:

```
"C" programme to greet the world.
The function "main" is the main
funtion.
Printf is the print function.
```

hello.c:

```
/*
"C" programme to greet the world.
*/
#include <stdio.h>
/*
The function "main" is the main
funtion.
*/
main()
{
/*
Printf is the print function.
*/
 printf("Hello, world!\n");
}
```

Figure 7.3.4: Mux file with multiplexed C and documentation, and files generated by demultiplexing - The Mux file contains explicit directives and generates both C programme and a file of comments alone. Some sectors generate date to both C output file "hello." and document file "hello.doc", and other sectors generate data only to the C file. In actual use, macros would be used to more smoothly integrate components, and hide the explicit Mux directives.

The only implementation of Mux is on Unix, but Mux itself in no way depends on it particularly. Moreover, while Mux was conceived with interprogramming in mind, it applies just as well to other cooperative strategies such as reliance on compilation to a common implementation base. It has been applied

in exploration for all of the purposes mentioned, and most extensively in implementation of another interprogramming tool described below.

INTERLANGUAGES

In practical interprogramming, there will clearly arise difficulties of incompatibility between data representations in use by otherwise communicating programming language environments. Such difficulties may be overcome by specific effort in either programme directly concerned, or by an indirect programme chosen specifically to expedite communication: an "interprogramme".

In the wide general case, of course, an interprogramme might always be necessary. In theory no common thread may ever be assumed, and even in practice the thread may often be surprisingly fine. However, there are reasons for staying within a language context doing whatever can be done to ease communication. Better comprehension of the operation in the "native" language might be one motivation, integrity of complex structure another. Moreover, there must be some concern about interprogramming itself: it cannot be arbitrarily free, in either pragmatic or linguistic costs.

To isolate interprogramming concerns yet limit linguistic costs to some extent, an interprogramme might use one of the languages already directly concerned. However, not all languages are well suited to this role. Complex representational bit twiddling is an intended capability of few languages, and - especially with a wide view of language - even the simplest representational manipulation is beyond many. Some languages might be easily extended, with a library of appropriate procedures for example, but this itself might be both a pragmatic and linguistic bother, and might seem totally inappropriate to some languages.

In at least some cases, then, an interprogramme needs a programming language particularly well suited: an "interlanguage". The language must allow easy access to and afford efficient manipulation of common implementations of common data structures used in the environment concerned. Many programming environments have a language that is somewhat appropriate to this task, if not a "high level" systems programming language then perhaps even an assembler language. BCPL, C, or Modula-2 are widely used portable systems programming languages that could be suitable as interlanguages, and there are many others - Appelbe and Hansen (1985) have compiled a wide survey. However, while interlanguage portability might assist overall portability, it must be remembered that the subjects of interprogrammes might themselves differ between implementations. So while the interprogramming idea may be portable, and while all the languages involved may be portable, any interprogrammes might well not be portable. This is both a weakness and strength: weakness because a portage must mean verification and possibly modification of interprogrammes; strength because without convergence of representation, interprogrammes at least isolate the divergence.

While systems programming languages can clearly do the job, they too have drawbacks. For while they may encompass the requisite capabilities, they might also encompass many others. In many potential ways, efficiency, ease of learning or of use, new portability, that extra baggage might seem a nuisance. Such

is the case for a new programming language specific by design for the interprogramme and for interprogramming. While diversity is problematic, innovation is still not only often appealing, but also of linguistic interest.

Several factors are of importance in design for a specialist interlanguage. Like a systems programming language, there should be easy access and efficient manipulation at the low levels of implementation. But an interlanguage is an adjunct to programming, and both ease of learning and ease of implementation are important, and with help from collaborating programmes an interprogramme should not need be very large itself, so a simple language is perhaps suggested. Flexibility is also an important issue. The low level details in differing environments might well differ: some architectures might provide swapping of bytes in a two-byte word, others may not; some architectures may provide both bit shift and bit "roll", others may not - examples abound. It would also be useful if whatever capabilities were offered in an interlanguage were also offered in direct form to such "systems programming languages" as might make use of them via, for instance, procedures or functions. These suggestions are open to debate, of course, and the difficult nature of language is such that no deterministic construction follows without question.

Languages especially for data conversion have been proposed and developed, but mostly address the application areas of file and data base management, and do not seem particularly appropriate for interprogramming. Sibley and Taylor (1973), for example, discuss a full and portable language for file conversion, and Shu, Housel, and Lum (1975) present a language with a high level of abstraction oriented about hierarchical data bases. Wolberg and Rafal (1978) offer a language expressly for textual conversion of data and programmes, more specific than Snobol or Icon, more general than a stream text editor, but still oriented at the interpretive and textual level. Stream editors like the Unix Sed (McMahon, 1979) are convenient in being oriented about familiar editor commands, and are enormously useful in simple cases, but seldom offer more sophistication than involving regular expression patterns. Awk (Aho, Kernighan, and Weiberger, 1979) offers only regular expression based patterns, but does also offers facilities for variables and simple control flow. It has been advocated as both a more specialised stream editor, and as an easily used programme generator (see van Wyk, 1986). However, Awk cannot do string substitution or complicated matching involving nesting, is still strictly character based, and is not implemented for efficiency.

In many ways the structural problems of data translation are similar to the structural problems of lexical analysis and parsing generally. Both are also situations where the efficiency of compiled implementation would often be worthwhile. Tools for building lexical analysers and parser semi-automatically are available, such as the Unix programmes Lex (Lesk, 1975) and Yacc (Johnson, 1975). Lex translates a specification based on sophisticated regular expressions to a finite automaton, and Yacc translates a specification based on a context free grammar to a push-down automaton. In both cases the automata are C programmes, and extra semantics must be written in the specifications in C. Data translators can indeed be built directly using these tools, but they are oriented to more general needs, and a programmer would have to do some repetitive low level work. Moreover, programmers who seldom use Lex and Yacc often find the detail of attention required bothersome anyway.

A tool to make Lex and Yacc more easily used in such programming is "Lys" (Biddle, 1987c). Lys (for Lex and Yacc Simplified) takes a programme specification and creates an executable programme in apparently one step, though in fact traversing several automatically. The programme specification uses a combination of Lex, Yacc, and C. The format is basically that of the middle or "rules" section of a Yacc programme, resembling a Backus-Naur Form grammar. A Yacc grammar, however, requires token names used for terminals, declared elsewhere, and defined in a separate lexical analyser. A Lys programme permits terminals to be specified as Lex regular expressions directly in the grammar. Lys does, like Yacc, still require the programmer to write the semantics of any grammar rules in C. However, it does provide a set of subroutines for building and using hashed symbol tables, a set of subroutines and macros for easily accessing parameters passed on programme entry, a facility for switching input and output connections, and a mechanism for logging the scanning process to help de-bugging. The symbol tables are actually used to store all strings, so also allowing easy string comparisons. For data translation, input can by default be taken from standard input or by name, and output similarly produced, but easy connections to names - files or tagged pipes, is also provided for. While Lex regular expressions are character oriented, direct interception of input and output is easily arranged, so temporary encoding can be done.

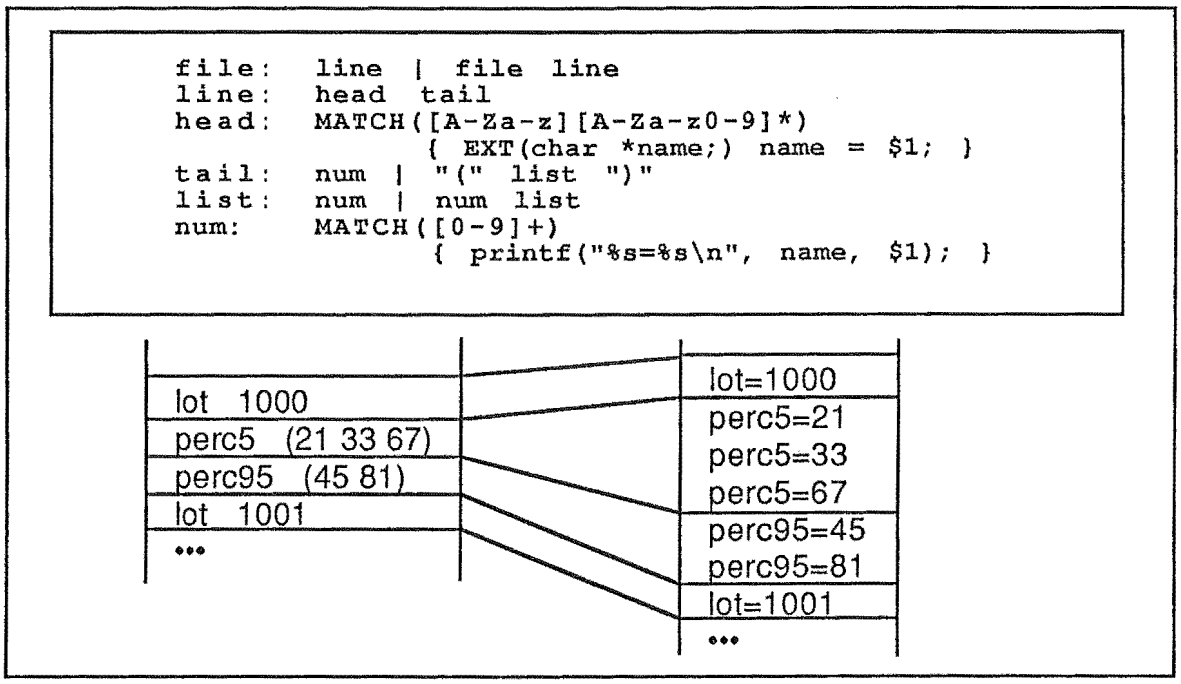

Figure 7.3.4: Lys programme for simple data translation - Names and either single numbers or parenthesis enclosed lists are input; names, "=", and single numbers are output. The programme is complete, and generates a working executable translator readfrom standard input and writing to standard output. Most of the programme is Yacc grammar, and includes the C code enclosed in braces. The "MATCH" directives generate Lex patterns and matching Yacc tokens. The "EXT" directive arranges it's argument to be external to the C code, so in this case declaring the variable in external scope. Like all Lys programmes, this is a Mux file: MATCH and EXT are predefined M4 macros expanding to Mux directives.

Lys is implemented by processing Lys directives in the specification with pre-defined M4 macros yielding a text then de-multiplexed by Mux. This results in some sectors becoming a Yacc specification,

others becoming a Lex specification, and yet others becoming both. The resulting files also need some postprocessing by a stream editor to fold multiple uses of the same regular expression. Lex and Yacc are then used to generate C programmes, and then they and the Lys library are compiled and loaded. All this is coordinated automatically by Make, and an executable data translator produced.

Lys still suffers from one problem of Lex and Yacc: the user really must be a reasonable C programmer. However, Lys certainly does ease the programming and run-time debugging, allows more compact and easily understandable programmes, and yields reasonably efficient data translation. Lys has proven successful in practice for both simple data translation programmes and for other less simple structural mappings, including a complete cross-assembler.

```
{< select main                    >}

M4  Macro  Definitions...

@define(EXT, `{< sector precode >}
$1 {< sector >}')
...
@define(FUNC, `{< sector code     >}
$1 {< sector >}')
...
@define(mn,0)
@define(MATCH,`@define(`MN',@incr(MN){< sector yaccdefs
{< sector
>}`MPAT'MN {< sector >}{< sector tokrules >}
$1     `MPAT'MN      {yylval=(int)tabput(lystab,yytext,yytext);return(`MPAT'MN);}
{< sector >}')

Mux  ordering  sequences...

{< sector main             >}          {< sector lexbits          >}
{< select yaccbits yacc.LYS >}         {<     select lexdefs      >}
{< select lexbits lex.LYS  >}          %%
{< select tokrules tok.LYS >}          {<     select lexrules     >}
{< sector >}                           [ \t\n]       ;
                                       #include "lex.tok.LYS"
{< sector yaccbits         >}.         {return(*yytext);}
#include "yacc.tok.LYS"                {< sector >}
{<     select yaccdefs     >}
{<     select yaccprecs    >}          {< sector code >}
%%                                     #include <stdio.h>}
{<     select yaccrules    >}          ...
%%                                     {<select precode >}
{<     select code         >}          ...
{< sector >}                           {< sector >)}

This  is  the  default  sector  in  effect  for  the  Lys  programme...
{< sector yaccrules >}
```

Figure 7.3.5: Mux prefix file for Lys - (slightly abbreviated) - The only complex M4 macro is "MATCH" which itself defines a new macro "MN" using the M4 built-in macro "incr" to each time return a higher value. "MN" is used to generate a unique token name beginning MPAT used then in both a Lex rule to return the token, and the containing Yacc rule. The Lex rule is first directed to a separate sector "tokrules" which is generated to a file and processed with Awk to remove duplicate lines in creating the Lex rule file "lex.tok.LYS", and to assign the same value to duplicate token in the Yacc/C adjustment file "yacc.tok.LYS".

This is one possibility for an interlanguage design, there are undoubtedly many others, and much innovation worthy of consideration. For instance, perhaps manipulation of bit strings might be approached in

a similar way to manipulation of character strings, and a Snobol or Icon like language might result. Whether or not interpretive translation is efficient enough would be a concern, but in many applications greater ease of programming might be more important. The role of the interprogramme in interprogramming is important: itself it enables some interprogramming; it facilitates more. But in how this essential flexibility can best be provided there are many language possibilities: from a language already directly involved, to a language chosen as an interlanguage, to a language designed as an interlanguage. Any precise choice itself, and any precise design itself, is another open question of programming linguistics. A diversity of interlanguages is not be something to necessarily be avoided. Scanner generators, stream editors, and simple transliteration programmes all have their place, and filling new places is simply evolution.

SECTION IV. CONCLUSION

To support programming language collaboration as outlined, interprogramming, there was a condition that supporting environments would need to provide sufficient basis for coordinating communication between language context openings. In practical application, the support level appropriate is the operating system, and many operating systems already address these concerns. In both the two large operating systems discussed, however, there is something left to be desired in support of interprogramming.

In the case of Primos, the difficulties stem from the design orientation about a single process supporting a whole terminal session. This design makes coordinating independent parallel programme envelopes very difficult. Indeed, it can even make coordinating independent serial programme envelopes difficult at times, for instance when a rogue programme interferes with the "command level" structure. (This does happen in practice, and the whole terminal session must be re-initialised.) That "command level" structure is an interesting approach to the particular need for temporary escape and later return to a programme, but is insufficient without extension for coordinating several programmes at once. When "phantom" terminal sessions are used to gain the parallel and protected structure of another process, the protection is in fact too great. While it is possible to write multi-process programmes in Primos, not being able to share open files or devices - particularly a control terminal - is a great liability. Where such problems can be borne, however, communications are possible through purpose-built facilities depending on the provision of file-associated semaphores. The method does work, though is perhaps less efficient than desirable. And while some transparency between such a messaging arrangement and ordinary file access is possible, that transparency is not provided by Primos itself between file access and device use - or even between use of different devices.

In summary, Primos unextended would make only a very primitive platform for supporting any general interprogramming approach in programming languages. The commitment to programme multiplicity is too weak, and the lack of generality in input/output access is also a hindering design.

The Unix situation is much more promising, though not perfect even so. The Unix process offers an appealing combination of protection and yet ability to share and communicate freely. The Unix pipe allows efficient inter-process communication and high transparency with other input/output. Much of the Unix design that lead to this is itself expressly oriented about interprogramming concerns, but the structure is not as free as might be wished. The importance of the process hierarchy is central, for instance, in sharing access to open files through "fork" and "exec", and makes non-hierarchical sharing difficult. The conventional use of the pipe by descendent processes in the "pipeline" is stricter than even a hierarchy requires, and more arbitrary structures can be built without changing the system. However, the pipeline structure does mean the lack of granularity control in input/output requires no care. In more general structures the granularity problem must be specifically catered for.

Unix already supports the interprogramming approach much more than Primos, where pipelines and even input/output redirection are unknown. And with the methods described, there are even more possibilities. However, it would be useful to make extensions for those methods to be assisted, with explicit

design for "named" pipes for example, and with better granularity control. There are many Unix variants, and the former - though not the latter - is a popular extension already.

The two operating systems discussed seem on opposite sides of a divide. In Primos, interprogramming is so cumbersome it probably is not worth pursuing. In Unix, interprogramming is already on offer in limited ways, and seems sufficiently easily improved to be worthwhile. This distinction is a lesson in operating system design. In order to generally cope with programming language diversity, operating systems should well address process coordination and communication: if not through multitasking and integrated message passing, then by whatever other able means devised.

It is tempting to speculate on operating system design based solely on this interprogramming approach. However, operating systems have various particular concerns that are not encompassed in the approach. One concern is inevitably efficiency, and it cannot be universally decided how much generality it is wise to pay for in that coin. Moreover, an operating system can itself be a linguistic context, and sweeping design on the basis of interprogramming needs is no more justifiable than in the programming language case. If programming language collaboration is sufficiently important in design of a particular operating system, the support should be provided. If nothing else is important at all, some network architecture could form a degenerate operating system. Interprogramming - what it will yield and what it will require - is an issue for consideration in the design of particular operating systems for particular purposes. If there is any imperative, it is that this approach has been insufficiently explored and merits specific attention.

And in a sufficiently facile interprogramming environment, needs and opportunities would arise for new programmes and programming languages between programmes and programming languages. This has already started in environments like Unix, and would become more significant with more commitment to interprogramming. At one time, operating systems provided one programme control language - specially provided for. Now there arises the possibility for many, each addressing particular concerns in connecting programmes together: so with Make and software building, and so with Rap and remote system manipulation. A desire for new language juxtapositions might once have lead to large new specially implemented combined systems. Now there arises the possibility for many combinations of existing languages and implementations to be used together: simply and easily multiplexed for use with Mux or programmes similar. Data translation used to be something done on large files by batch programmes. In support of increased direct traffic between contexts with differing data representations, there arises the need for flexible but efficient stream oriented translators: stream editors like Sed, and more easily used lexical analysis and parsing software generators like Lys. There are undoubtedly more possibilities for design in these areas, probably sufficient scope and need for new kinds of software altogether.

CONCLUSIONS

From INTRODUCTION:

Over all, this exploration of solutions to the problems of programming language diversity is structured with three divisions progressively narrowing in scope.

First is a consideration of the possibilities for reasoning to convergence in programming languages, thereby eliminating the problems of diversity by eliminating the diversity: integration at the roots of diversity. Central here is the question of programming language "quality" and how it might be evaluated. These questions are extraordinarily deep, and conclusions can really only underline the difficulty and importance of the problem. In these discussions scope is generally restricted to matters directly involving computing. As matters seem to lead away to other experience and more general enquiry, this is somewhat arbitrary and perhaps rather unsatisfying. Within the practical context, however, simply establishing the direction of difficulties is sufficient.

Second is a survey of established computing practices and techniques that in some way approach solution to the diversity problems. This a reminder that much that is impossible to totally resolve can still managed with tolerably. The approaches examined are grouped into those seeking "compromise": reducing different languages to one, and those seeking "cooperation": ways of using different languages together. Often the diversity problem has been only a minor or unstated concern of formal developments and studies, so some conclusions can be only informally drawn. Many strategies still have definite advantages in certain circumstances, however, and several deserve to continue being applied. But most strategies also have significant drawbacks that do keep their usefulness limited. Even so, some cooperative practices appear on examination to suggest further development of a more general strategy may be worthwhile.

Third is the proposal for a new general strategy for addressing and tackling the diversity problem. The philosophical problems realised in the first part of the study and the practical directions suggested by the second section do not quite seem in collision. Indeed it seems possible to practically evade problems of diversity as much as any particular programming language design permits, while avoiding any necessary implication of impossible universality. So such success must necessarily be limited, but some increased success is possible. The approach involves minimally collaborative design in programming languages, and necessary support from environments exterior to them. Though grounded in applicable practice, these proposals do not follow through in requiring specific action - there remains much choice. These are design principles for languages and support systems, and could be evolved toward. However, several programming languages are explored, and opportunities for useful access described or modifications suggested. To practically examine the question of the exterior support, two operating systems are also discussed, and the requirements of necessary design established. The importance of new software both to assist and take advantage of this "programming between programming languages" is illustrated with some new programming tools.

So more exploration and experience are indicated, and detailed study for particular programming languages and operating systems design would then be of most concern, as well as more attention to particular questions of performance and efficiency. Such particular work is outside the current scope, as indeed are several other directions suggested for future investigation. Explorations in representation diversity in other fields may deserve investigation next, and the balance of subject matter could be important as well as enticing. The contributions of this study are the mapping out of the vast area of this important subject, the review of relevant past practice, the development of a general strategy for progress, and the directions for its practical pursuit.

CHAPTER VIII
CONCLUSIONS

This consideration of solutions to the problems of programming language diversity begins with a wide view but ends with a narrow view. This progression is from discussion of the persistent problem of understanding language quality, through survey of related practical measures, to proposal and exploration of a new strategy. It is a course from a formidable puzzle to a reasonable solution. Along this course there has been much resolution, but there has also been suggestion of directions for further exploration. Because the scale is large, resolving the issues into a structure is itself useful, so the major components of that structure are worth looking at again, with attention to possibilities for future further work. The study has been divided into parts looking at convergence, coping, and collaboration between programming languages. Each part is distinct in the character of resolution and of suggestions for further investigation: there is a clear progression from the philosophical, through the documentary, to the exploratory.

SECTION I. CONVERGING

The starting point in examining possible approaches to resolving the diversity problem was at the fundamental level. The diversity arose at least partially for historical reasons and through inexperience, so the possibility of reasoned convergence could not be immediately ruled out. However, it quickly became clear that such resolution involved understanding the very slippery question of language "quality". In discussion of the difficult questions consequently raised, the widest experience of language seemed applicable. An initially operational view was taken, and language convergence discussed in implementation and then in application. In attempting to reason about application, several approaches were inspected: simple linguistic precedence, mathematical theory, psychological experiment, and empirical analysis. While some progress seemed reasonable in understanding quality in language implementation, the human factor in language application lead to great difficulty in reasoning further. Indeed not only does the question lead to wider philosophical enquiry, but the search for resolution at that level meets with difficulties so severe as to conceivably transcend solution. Arguments were outlined to link problems of reasoning about programming language with problems of reasoning about natural language, then those wider difficulties were surveyed very briefly. There is interaction between objectivity and subjectivity, between language and context, and a very intractable relativity results. In conclusion there was only really the underlining of the enigmatic and persistent nature of the problem, suggesting that reasoned convergence of programming languages is not a practical possibility.

Further Directions

The deepest difficulties in understanding the nature of programming language quality apply also in understanding the nature of any language quality. If this consideration were to be further pursued directly, relationships between programming language and other kinds of "language" would be of great interest and

importance. On a theoretical level, the unity of nature of programming and other language could be specifically investigated. There are many theories about the origin and structure of natural languages, and it it would be interesting to see if their application to programming languages yielded new insight. In particular, it would be useful to more formally relate the difficulties about linguistic relativity in programming and natural language - or indeed the difficulties about linguistic universals. While much more study has been done on natural language, it might also be the case that programming language is of interest to linguistics generally. Compared to natural languages, the field work in gathering raw material would be quite easy. Compared to other special purpose languages, programming language probably presents a very wide span in application and sophistication. Some aspects of programming language must be regarded specially by linguistics, however, particularly those aspects strongly affected by the imperatives of implementation. The mathematics of formal language theory pioneered by Chomsky was in descriptive exploration of language, yet those same formalisms have now become prescriptive in programming language design. Simple finite state and context free grammars were clearly shown inadequate for natural language, yet are now the implementation backbone of programming language. But even this phenomenon itself should be of interest in linguistics. Formally, it does illustrate how language of greater computational and so generative power can be built structured largely on a framework and hierarchy of simpler grammars. Less formally, programming language could show how idiom, humour, and cultural influence of all kinds can survive in even the most rigorous setting.

While the problems of understanding language quality are deep, within all language communities it is usual for common practices to evolve as new circumstances arise. An important difference between programming and natural languages, however, is that individual programmers and even communities of programmers have less ability to change programming languages by themselves. On a practical level, then, more work could be done in studying existing practice in programming languages in order to facilitate evolution there. This is mainly a question of those in a position to change languages keeping better in touch with the programmers who use them on a day-to-day basis. Even so, there are reasons why such an approach is unlikely to match the natural language freedom. There are the technical reasons that make changing a programming language a weighty matter: existing programmes and other material are long-lived, and centralised technical work in creating and modifying the implementing software is difficult. There is also the pragmatic question of whether those implementing the language agree with those using the language. Such reasons cannot impede totally, though, as users can and do effectively change languages with pre-processors, and even new complete implementations arise when feelings are sufficiently strong.

Knowledge about common practices is also interesting from the viewpoint of pedagogy ("why is this done? why not?") as well as from concern for particular software engineering practice. The use of "nosy software" to automatically collect information on practice is an open possibility at several levels, and of potential promise in both language application and implementation. Analysis in general would seem quite difficult, but more straightforward if done sufficiently locally. More work in this direction, both in techniques for data capture and for subsequent analysis, should be worthwhile. And generally, of course, there are all the avenues of language analysis to be pursued further: study of precedents, mathematics, psychology, and all contextually relevant fileds. If in practice representation does matter, study of the effect s of representation anywhere can only help in greater understanding of the linguistic factor.

Beyond any theoretical or practical analysis is continuing opportunity for exploration and innovation in programming language design. With so weak an argument for reasoned convergence, perhaps computing should look more openly at new possibilities for expression. If the creative arts are a reasonable context for this branch of computing, then design so far has been surprisingly conservative. In the traditional context of practical computing, this attitude may appear frivolous - but there are other contexts. There is no reason even to rule out computer programmes and programming languages as art objects, nor programming as performance art. And without strong arguments for reasoned convergence, there remains the possibility that exploration in other contexts may lead to illumination in programming language design for even the most traditional computing.

SECTION II. COPING

The inability to reason to convergence implied that diversity was something to be coped with as best as possible, rather than solved outright. In the survey of practical computing techniques concerning language diversity, several appear to usefully limit the problems, if not approach solution. So while persistent and difficult the problems may be, coping is possible to some extent. Two general methods of coping were seen, those resolving diversity through compromise and new design, and those accepting diverse designs but seeking some cooperation. As compromise were discussed standardisation, minimal design, multiple design, extensibility, and abstract or "pseudo" language. As cooperation were discussed direct translation, language subsets and supersets, flexibility in definition, reliance on a common implementation, and coordination with programme control. Many methods discussed were not in fact conceived with regard to diversity problems at all, though in particular situations all the approaches have merit.

In compromise, the general merit is that of standardisation: helpful simplification when there is already some agreement. Where there is agreement on a sufficiently powerful kernel of language design, minimisation will make portability easier, and encourage wide distribution. And when there is clear agreement on factors from several languages all needing to be on offer, a multiple language can match that agreement with close integration otherwise unattainable. Where the agreement is only on a minimal kernel but more flexibility is needed to make the language really usable, then some form of extensibility is a very useful choice - especially if the additional capability required itself needs to be explored. And where committing to an actual language is not regarded important, but where there is consensus on the general programming paradigm anyway, some form of abstract language can there be very helpful, especially at the design level.

In cooperation, the general merit is allowance of some choice in language, yet retaining compatibility across such choice. Translation is there the ultimate answer, and though automatic translation can rarely produce good results linguistically, in the situation of moving one-way to a new language even this cumbersome strategy is still reasonable. Where languages can be arranged as subsets or supersets of one-another, then easier translation and even more immediate inclusion are possible - but only along very strict

lines. Where the necessary differences are only lexical, language definition itself can be flexible, and much compatibility can be either built in to language implementation. That by itself can span little, but where a common implementation method spans several languages, the compatibility is great as long as the base is assured. Where implementations of programmes in different languages can be coordinated to work together, then there is potential for language spanning to also be portable , though the coordination structure is usually limited.

Yet a few are not only serviceable in a restricted way already, but are open to further development. Particularly promising seemed the approaches in coping by cooperating, especially reliance on a common implementation and using programme coordination, so inviting the exploration of more explicit collaboration.

Further Directions

In this look at practical techniques, the attempt was to include relevant general practice. As a separate venture, however, it would be interesting to attempt to catalogue more specific situations of coping with programming language diversity. With greater breadth and depth of coverage, the taxonomy should be reviewed, however. Even the simple distinction drawn here between compromise and cooperation is not always clear, and several approaches discussed contain aspects of both. A more detailed survey would be useful as a historical perspective, should help in corroborating the direction seen here toward cooperation and collaboration, and might of course uncover new directions unimagined. More detail would be worthwhile even for certain specific cases, indeed covering the full breadth of the field would be very difficult. There are many programming languages in the world, and more every day. Many are ill documented, and especially so the roots of their foundation.

SECTION III. COLLABORATING

In outlining the collaborative "interprogramming" strategy for approaching solution to the diversity problems, practical orientation narrows to focus. By explicitly acknowledging any arbitrary exterior to a programming language context, the strategy provides the opportunity for connection and communication between different such contexts. And, programming languages so providing, a supporting environment could then realise such connection and communication. The approach is essentially one of building at least minimal facilities that ensure the ability of arbitrary escape from, and recourse to, any particular language. In theory, the provision of opening facilities inside languages seems no difficulty, and realisation outside in supporting environments seems sufficiently reasonable. The fundamental technical problem of programming language diversity is resolving one working practical programme where components, through fortune or desire, are in different programming languages. To the extent that programming language diversity is a technical problem, interprogramming is a general solution - because it enables flexible programming between languages.

In practice, the essence of the new provisions requires detailed design and implementation. The approach generally depends on individual languages being provided with explicit openings in their context. However, the fundamental philosophical problem of programming language diversity is that one cannot rule on programming language quality. So in any particular case, there is still wide linguistic latitude that must be afforded. In exploration, however, of how context openings might be integrated into well known language designs, several specific languages were examined: Fortran, Pascal, C, Icon, Prolog, and some less "linguistic" software as well. And in exploration of how such context openings might be connected for communication, two operating systems, Primos and Unix, were also examined in detail. Most programming languages provide, or almost provide, minimally adequate openings in context, particularly in conjunction with input/output. Many languages further provide opportunity for larger schemes integrating the "opening" method into the fabric of the overall language. Actual design in any of the popular languages described is a pragmatic matter of negotiation, implementation, and acceptance in their respective communities. In detail, language design needs to be more precise in allowing control of openings to the programmer - this in input/output and related granularity issues, and also in some cases of evaluation order indeterminism. In principle, however, there seem few problems in providing at least minimally sufficient opening facilities for interprogramming.

The operating system situation is not quite so clear. The two main requirements to support the interprogramming are some form of multiple envelope provision, and some form of communication between such envelopes. In practice, these would commonly arise in multitasking and message passing. Both the operating systems discussed are modern, popular, and were designed for wide general purpose use. Primos, however, was just short of the level of commitment really needed to support interprogramming. It is a multiprogramming system, and protects different terminal session environments from one another, but does not provide a sufficient structure beneath that level. Even within what primitive structure there is, communications are not well catered for - though without any barrier they can be still accomplished. Using the multiprogramming level itself, programme components in different "phantoms" do provide sufficient isolation and protection for separate language environments. Communication is then possible, but access to common files and devices, like the control terminal, is not. Unix does provide a sufficient frame to support interprogramming. While multiprogramming, it is also multitasking within the same structure, and the process is isolated enough for an individual programming language environment, yet also provided with possibilities for communication through "pipes" and sharing of file and device access. Processes are structured together hierarchically, though, and communications are restricted somewhat by this structure. It is possible to provide communication across the structure, however, merely by using a preparation programme and an alternative input/output subroutine library. Unix does provide a limited kind of interprogramming through the "pipeline" process linkage already, and this approach simply extends it. The more limited linear structure does avoid problems of coordination because input/output granularity is not well provided, but that problem too can be dodged. If Unix were to be extended, more arbitrary access across process hierarchies and better input/output granularity would both be changes useful for interprogramming. The operating system support for interprogramming is the critical level. The approach is workable, but operating systems really do need multitasking, preferably with memory protection, and easily controlled

preferably integrated with general input/output. Though Primos was designed after Unix, the Primos design is not quite enough and the Unix design is just sufficient.

Interprogramming directly involves programming languages, and must rely on operating system support. But there is one other factor involved: new programming languages to facilitate the interprogramming. These facilities build on the interprogramming as well, and would become more important in a growing framework. Three kinds of such language were discussed, though others may well arise. "Meta-programming" languages were distinguished as languages where other programmes are component parts. Control languages are the established member of this group, but others are developing. The Unix "Make" facility was looked at as a very specific example addressing programme maintenance, and a new language "Rap" was introduced as offering a meta-programming facility to couple programmes on one computer system with another system. "Interprogramme" data translators were discussed as programmes to translate interprogramming data passing between programming languages. Conventional programming languages can be used for this service, but the specialised area of discourse was seen as suggesting special-purpose "interlanguages". Ordinary stream editors are simple interlanguages, with pattern matching based on regular expressions, and for maximum power full languages like Icon would be appropriate. A new proposal, "Lys" spanned these levels of recognition power in one compact unit, combining the regular expression matching of Lex, the context free grammar capability of Yacc, and the overall flexibility of C. This was an attractive combination, and suitable to many programming language situations because so many languages structures are similarly composed. Programming language "multiplexing" was a less immediate need in making interprogramming work, but was an interesting example of the linguistic possibilities that can flow with improved language cooperation. Where interprogramming provides the operational connection between programmes in differing languages, multiplexing can offer analogous textual juxtaposition. A general facility, "Mux", was proposed as a general purpose multiplexor, applicable to any set of textual representations. Mux allows distinct text files to be described with interleaved and possibly overlapping sectors, selected in controlled order. The Mux directive meta-notation is simple and designed for minimal interference with other notations involved. Even so, more harmonious multiplexing results from structures tailored to the application, and Mux provides integral access to macro-processors for a general such service.

Further Directions

The direction suggested by these explorations is toward more practical implementation. Most important is the operating system level, where the necessary support of interprogramming must be founded. Existing operating systems should be examined for their suitability to provide the necessary coordination and communication facilities. Where systems do provide a sufficient basis, as Unix does, actual design of the facilities could be done to best fit in with the operating system context, and constructed on the existing basis, as was described for Unix. Where systems do not really provide enough grounding, the reasons why should be determined, and if possible the design decisions that lead to that situation examined. It is important to understand where such design is justified, to identify better the bounds of interprogramming rationality. Where the design does not seem counter to interprogramming, but nevertheless just insufficient, extensions

to the system should be designed and perhaps constructed. Operatings systems are large and often proprietary, however, and actual extension possibilities are few - if only because of the commercial and legal restrictions. Beyond extending existing operating systems, design and construction of a new operating system could also be considered expressly to support the interprogramming approach. Such a system would provide user multitasking with memory protection, file and device sharing between processes, and inter-process communication with granularity control integrated with file and device input/output. Some aspects of operating system design are not directly affected by the issues involved, low level file system design for example, so some existing software could be used. However, some general aspects of operating systems are indirectly involved. Scheduling algorithms, for instance, should be tailorable to user multitasking within multiprogramming, as those described by Larmouth and Henry are.

The next step in realising practical interprogramming should be implementation of language alterations and extensions for context opening. As described, openings could be implicit or explicit. Initially, however, it would be prudent to avoid changing languages explicitly, or at least to incorporate only additive explicit changes. The language openings described here were intended as exploration and illustration of possible approaches. In practice, pragmatic details will have to be given more attention, especially if wide acceptance is sought, and these considerations would likely limit more ambitious language experimentation. As detailed in the examples, the important issues will be access to language components, granularity control, generality through integration with existing openings, and maintenance of harmonious design. However, the standardisation issue cannot be ignored long. While the use of interprogramme translators and coordinators will enable interprogramming in important cases, it would be significantly easier with standards. Low level standards are desirable, of course, but concern for implementation efficiency usually makes them difficult to achieve - and higher level structures still vary. Input/output facilities provide the most successful standards now, and greater standardisation in external manifestation of data structures should be explored. Unfortunately, because this is again a human application concern, the level for standardisation should perhaps be between application and implementation - this matter needs investigation.

With sufficient grounding at the operating system level for coordination and communication, and with programming language design incorporating sufficient openings in context, interprogramming will be possible. At that point some attention should be given to the last factor of interprogramming discussed, programming languages between programming languages. There is much scope for new work. In each of the three particular categories that were discussed, there is room for improvement and for new languages. In "meta-programming", programming is done with whole programmes in other languages as components. Even in the most traditional kind, the control language, there are opportunities for control languages for beginners, for languages allowing more efficient implementation that the usual interpretation, for logic oriented Prolog-like control languages, and so on. There are more sophisticated versions of Make available now, coping with libraries and more complex dependencies. But there there might also be such systems better tailored to particular users: again a beginner's model for instance, or perhaps a production model with automatic keeping of records and backups of software changes. Rap is a very useful tool for manipulating remote systems automatically, and connecting local programmes. But with more sophisticated control structures for programmed iteration, it could allow more flexible connection with programmes, and require

less preparation done on remote systems. Data translation "interlanguages" would be quite important in practical interprogramming between languages wide apart in representations of data. Between simple editing facilities, and full programming languages, the Lys proposal appears focussed at a most useful level for creating efficient data translation programmes. While offering users more application services than Lex or Yacc, however, Lys can still be difficult to use, especially in syntax error recovery and debugging . It inherits these faults directly from its bases, all three having those same disadvantages. A language with similar offerings to Lys but with a more robust implementation would be appreciated, especially by users unknowledgeable or uninterested in the wider excitement of Lex, Yacc, or C. And without or until significant standardisation in context opening design, all interprogramming must rely on such translation facilities, or use limited facilities in input/output together with ad-hoc translation. The current implementation of Lys uses Mux, and Mux is serving well as a prototyping facility. Language "multiplexors" like Mux show a more unusual direction generally. Where languages can be combined operationally, it may make sense to combine them textually, and Mux makes this an easy possibility. Mux allows general multiplexing, however, and while thus enabling easy prototyping, it does mean it is limited - even with use of a flexible macro processor - in allowing multiplexing very specifically tailored to the languages involved. New directions could explore more particular language multiplexing, suiting the exact syntax to the particular languages involved. An intermediate step might simply involve new "front end" software for Mux, instead of the usual macro processors. There is much potential here, and the possibilities for collage programming would be fascinating to explore further.

SECTION IV. CONCLUSION

The three parts of this study are quite different in character, represent different kinds of work, and result in different contributions.

Part one, "Converging", is an argument against the possibility of reasoning to convergence in programming language design. The difficulty in tackling the questions involved is the truly vast scope of relevant material. It is surprisingly easy to arrive at such questions in computing without realising the enormous force that other subjects exert on the answers. The relevant computing topics involved language implementation, programming theory, programming psychology and empirical analysis. But topics also relevant involved art, linguistics, and philosophy. Coming to terms with such a wide range of material, much of it with vast and complex literatures, is not easy. However, the most significant points from all these areas are covered in the argument, and result in substantial theoretical doubt about any possibility of reasoning to convergence in programming language design. While programming languages are formal computing models, they are also informal representation palettes for human thought and communication. The former aspect may indeed lead to reasoned convergence in scope either narrow or wide. The latter aspect allows very little reasoning, and through recursive and twisted complications of linguistic relativity, may allow none at all. This is an area for continuing research, but it is research at the perplexing edge of psychology. This leaves programming language design subject to the same difficulties in precise analysis that are characteristic of the arts. This is not generally understood in computing, and deserves to have more impact.

Part two, "Coping", is a survey of programming and programming language practices that have assisted in coping with the diversity. This involved much programming language history, and linking together many designs and practices otherwise little connected. Examples were chosen from well known languages and situations where possible, but the topic itself has not previously been surveyed. Though many of the approaches discussed were not conceived with tackling the diversity problem as a main goal, most of them do still offer advantages in particular circumstances as outlined. Moreover, when assembled together, the approaches also suggest a more general possibility, evolving from some established methods and encompassing others. Despite the earlier argument doubting the ability to reason to convergence, the result here does offer practical advice. For diversity problems in particular situations, there are particular approaches to solution recommended. For more general situations, a cooperative approach offering the advantages of common implementation and of programme coordination could be further developed.

Part three, "Collaborating", is an exploration of that possibility of explicit design for collaborating between programming languages, so to generally tackle the practical problems of diversity. This "interprogramming" design is based both in operating systems and in programming languages. The operating system role required investigation of potential facilities for coordination and communication between programmes, and favoured user multitasking and explicit message passing - though some choice was possible in detail. The programming language role required study of several possibilities for opening the language context. Allowance for great latitude there was an important part of the approach, so many designs were acceptable, involving both explicit and only implicit language change. Several particular languages were studied in detail, and many variations on the "opening" theme discussed. It appeared that most languages would permit useful opening with relatively minor adjustment, and were capable of encompassing more complete integration as well. Language design must ensure, however, to afford programmers the necessary control in using openings, and should seek standardisation in context opening communication and coordination where possible. Two operating systems were also studied in detail, and exploratory programming done to ascertain interprogramming possibilities in practice. The Primos experience resulted in deciding against its suitability in particular, and in general for more flexible use of processes and more transparency between different kinds of input/output. The Unix experience resulted in deciding it was sufficiently suitable if used with the communications techniques described, but that less hierarchically oriented structures would be more useful, as would better input/output granularity control. The key to collaborating between programming languages to overcome the technical problems of diversity is in operating system design. In future design, this lesson should be remembered.

The last component involved in examining interprogramming was the exploration of new kinds of languages between languages. A language for controlling remote systems, "Rap", was designed and developed by students as illustration of "meta-programming" allowing use of other programmes as components. As an example of an "interlanguage" for translating data streams between languages, "Lys" was designed and developed as a very compact and efficient translation language based on the scanner generator Lex and the parser generator Yacc. And a new kind of software component - a general programming language multiplexor - "Mux", was also designed and developed. While Mux can be used to enable textual interweaving of separate languages for any reason desired, the potential for dramatically

representing juxtaposition between operationally communicating programme parts in differing languages is exciting. Mux, Lys, and Rap are all implemented as practical software, and have been used since their development in implementation of other unrelated projects.

Overall, this study has required much investigation over the wide areas of programming, programming language design, and context beyond, in attempting to understand the nature of programming language quality. It has involved examination of programming and programming language history, and correlating experiences in coping with programming language diversity. It incorporates a new proposal, subsequent analysis of programming language and operating system implications, detailed study and practical exploration with common systems, and design and development of new software. This work shows firstly why programming language convergence cannot be directly reasoned to. It then shows where, and under what conditions, strategies for coping with programming language diversity have succeeded and failed. Finally, it shows how the more successful cooperative strategies can be generalised with minimal collaborative design in both programming languages and operating systems. Programming language diversity is a large and important topic, and has been seldom directly addressed. This study has covered the significant breadth of the topic, and presented directions for real progress.

Programming languages are at the centre of practical computer programming activity. In design of programming languages, however, success has been so great and so diverse that the diversity itself has become an important practical problem. It is a tricky problem because divergence cannot be seen as wholly bad. It is not clear that reasoned convergence is possible. It is not clear that if it were possible it would be desirable.

Practice shows that where solutions to the problems of diversity cannot be approached through compromise, they may still be approached through cooperation. Such cooperation can be planned for and provided for explicitly in the design of programming languages and supporting environments. Such planning and provision is not too difficult, can be adapted to existing practice, and can be worked into new design. It can yield an ability to programme between programming languages.

Diversity cannot be denied, need not defied, but may be spanned.

REFERENCES

REFERENCES

Addyman, A.M. (1981) A Draft Proposal For Pascal. ACM Sigplan Notices, 15:4.

Adobe Systems Incorporated (1985) PostScript Language Reference Manual. Reading Massachusetts, Addison-Wesley.

Aho, A.V. (1980) Translator Writing Systems: Where Do They Now Stand? Computer, 13:8, pp.9-14.

Aho, A.V., Kernighan, B.W., Weinberger, P.J. (1979) Awk - A Pattern Scanning And Processing Language. In Leffler, Joy, and McKusick (Eds.) (1983).

Albrecht, P.F., Garrison, P.E., Graham, S.L., Hyerle, R.H., IP, P., Krieg-Brueckner, B. (1980) Source To Source Translation: Ada To Pascal And Pascal To Ada. ACM Sigplan Notices, 15:11, pp.183-193.

Allan, R., Irwin, W., Chong K.F., Leong Y.H., Biddle, R.L. (1987) Rap(1): Rapport Between Systems. In "Cantuar" Unix Programmers Manual. Christchurch, University of Canterbury Computer Science Department. [See Appendix]

American Telephone & Telegraph (1986) The System V Interface Definition. Indianapolis Indiana, AT&T.

Andrews, G.R. (1982a) Distributed Programming Languages. ACM 82 Conference Proceedings, pp.113-117. New York, Association For Computing Machinery.

Andrews, G.R. (1982b) The Distributed Programming Language SR - Mechanisms, Design, And Implementation. Software - Practice And Experience, 12, pp.719-753.

Andrews, G.R., Schneider, F.B. (1983) Concepts And Notations For Concurrent Programming. ACM Computing Surveys, 15:1, pp.3-43.

Ansi (1976) American National Standards Programming Language Pl/I. New York, American National Standards Institute.

Ansi (1982) Intra Language Compatibility Guideline. ACM Sigplan Notices, 17:7, pp.2-4.

Ansi (1987) IEEE Standard For Binary Floating Point Arithmetic. ACM Sigplan Notices, 22:2, pp.7-18.

Appelbe, W.F., Hansen, K. (1985) A Survey Of Systems Programming Languages: Concepts And Facilities. Software - Practice And Experience, 15, pp.169-190.

Apple Computer (1987) Macintosh HyperCard User's Guide. Cupertino California, Apple Computer Inc.

Arisawa, M., Iuchi, M. (1979) Fortran + Pre Processor = Utopia 84. ACM Sigplan Notices, 14:1, pp.12-15.

Arnheim, R. (1969) Visual Thinking. Berkely, University Of California Press.

Asteroff, J. (1987) Paralanguage In Electronic Mail. Ph.D. Thesis, Columbia University, New York.

Austin, J.L. (1962) How To Do Things With Words. Cambridge Massachusetts, Harvard University Press.

Ayer, A.J. (1971) Language, Truth And Logic. Harmondsworth Middlesex, Penguin Books. [Originally 1936]

Ayer, A.J. (1984) Philosophy In The Twentieth Century. Hemel Hemstead Hertfordshire, Unwin. [Originally 1982]

Bach, M.J. (1986) The Design Of The Unix Operating System. Englewood Cliffs New Jersey, Prentice-Hall.

Backus, J. (1978a) Can Programming Be Liberated From The Von Neumann Style? Communications Of The ACM, 21:8, pp.613-641.

Backus, J. (1978b) The History Of Fortran I, II, And III. ACM Sigplan Notices, 13:8, pp.165-180.

Baker, B.S. (1977) An Algorithm For Structuring Flowgraphs. Journal Of The ACM, 24:1, pp.98-120.

Balzer, R.M. (1971) Ports - A Method For Dynamic Interprogram Communication And Job Control. In Proceedings Of The AFIPS Spring Joint Computer Conference. Arlington Virginia, AFIPS Press.

Barron, D.W., Buxton, J.N., Hartley, D.F., Nixon, E., Strachey, C. (1963) The Main Features of CPL. The Computer Journal, 6:2, ppp.134-143.

Barron, D.W., Jackson, I.R. (1972) The Evolution Of Job Control Languages. Software - Practice And Experience, 2, pp.143-164.

Beckman, A. (1977) Comments Considered Harmful. ACM Sigplan Notices, 12:4, pp.94-96.

Beech, D. (1982) Modularity Of Computer Languages. Software - Practice And Experience, 12, pp.929-958.

Bell Telephone Laboratories (1979) Unix Time-Sharing System: Unix Programmer's Manual. Seventh Edition. Murray Hill New Jersey, Bell Telephone Laboratories.

Belpaire, G. (1975) Synchronisation: Is A Synthesis Of The Problems Possible? ACM Operating Systems Review, 9:3, pp.3-10.

Bemer, R.W. (1969) A Politco-Social History Of Algol. Annual Review Of Automatic Programming, 5, pp.151-238.

Berlin, B., Kay, P. (1969) Basic Color Terms: Their Universality And Evolution. Berkeley, University Of California.

Beth, E.W., Piaget, J. (1966) [W. Mays (Tr.)] Mathematical Epistemology And Psychology. Dordrecht Holland, D. Reidel.

Bickerton, D. (1973) The Nature Of A Creole Continuum. Language, 49, 640-670.

Biddle, R.L. (1983a) Relations In Programming Language Data Structuring. Australian Computer Science Communications, 5:1, pp.323-327.

Biddle, R.L. (1983b) Implementation Independent Multiple Language Programming. Proceedings Of The First Australian Computer Engineering Symposium, Newcastle New South Wales, The University Of Newcastle.

Biddle, R.L. (1984) What Can We Learn From Our Errors: An Empirical Look At Syntax Errors In Students Computer Programs. In Hughes, J. (Ed.) Computers And Educations: Dreams And Reality, Broadway New South Wales, Computer Education Group Of New South Wales, pp.313-315.

Biddle, R.L. (1987a) Ipio(3): An Inter-Programme Communication I/O Library. In "Cantuar" Unix Programmers Manual. Christchurch, University of Canterbury Computer Science Department. [See Appendix]

Biddle, R.L. (1987b) Mux(1): A Language Multiplexor. In "Cantuar" Unix Programmers Manual. Christchurch, University of Canterbury Computer Science Department. [See Appendix]

Biddle, R.L. (1987c) Lys(1): Lex, Yacc, Simplified. In "Cantuar" Unix Programmers Manual. Christchurch, University of Canterbury Computer Science Department. [See Appendix]

Bishop, J.M. (1979) On Publication Pascal. Software: Practice And Experience, 9, pp.711-717.

Biermann, A.W., Ballard, B.W. (1980) Toward Natural Language Computation. American Journal Of Computational Linguistics, 6:2, pp.71-80.

Birtwistle, G.M. (1979) Demos - A System For Discrete Event Modelling On Simula. Basingstoke London, Macmillan.

Birtwistle, G.M., Dahl O.-J., Myrhaug, B., Nygaard, K. (1973) Simula Begin. Lund, Studentlitteratur; Philadelphia, Auerbach.

Blair, G.S., Mariani, J.A., Shepard, W.D. (1983) A Practical Extension To Unix For Interprocess Communication. Software - Practice And Experience, 13, pp.45-58.

Bobillier, P.A., Kahan, B.C., Probst, A.R. (1976) Simulation With GPSS And GPSS V. Englewood Cliff New Jersey, Prentice-Hall.

Boden, M. (1977) Artificial Intelligence And Natural Man. Brighton, Harvester Press.

Bohm, C., Jacopini, G. (1966) Flow Diagrams, Turing Machines, And Languages With Only Two Formation Rules. Communications Of The ACM, 9:5, pp.366-371.

Bossi, A., Cocco, N., Dulli, S. (1982) Modular Decomposition Of Ada Into A Hierarchy Of Sublanguages. Ada Letters, 2:6, pp.53-58.

Bourne, S.R. (1978) The Unix Shell. The Bell System Technical Journal, 57:6:2, pp.1971-1990.

Brainerd, W. (Ed.) (1978) Fortran 77. Communications Of The ACM, 21:10. pp.806-820.

Brinch Hansen (1978) Distributed Processes, A Concurrent Programming Concept. Communications Of The ACM, 21:11, pp.934-941.

Brooks, R.E. (1980) Studying Programmer Behaviour Experimentally: The Problems Of Proper Methodology. Communications Of The ACM, 23:4, pp.207-213.

Brown, G.H. (1984) Colin McCahon, Artist. Wellington, Reed.

Brown, P.J. (1967) The ML/I Macro Processor. Communications Of The ACM, 10:10, pp.618-623.

Brown, P.J. (1972) Levels Of Language For Portable Software. Communications Of The ACM, 15:12, pp.1059-1062.

Brown, P.J. (1977) Software Portability: An Advanced Course. Cambridge, Cambridge University Press.

Brown, P.J. (1980) Supermac: A Macro Facility That Can Be Added To Existing Compilers. Software - Practice And Experience, 10, pp.431-434.

Brown, P.J., Ogden, J.A. (1983) The Supermac Macro Processor In Pascal. Software - Practice And Experience, 13, pp.295-304.

Buck, D.L. (1984) Reader's Guide To The 1984 /usr/group Standard. Santa Clara California, /usr/group.

Buckley, B. (1987) ASCII vs UNIX. Australian Unix Systems User Group Newsletter, 7-6, pp.8-11

Burley, J.C., Burns, E., Forbes, J., Lamb, S., Landy, A. (1985) Programmer's Guide To Bind And EPFs, Revision 19.4. Natick Massachusetts, Prime Computer.

Burnham, J. (1973) The Structure Of Art. Revised Edition. New York, George Braziller.

Burroughs (1983) Linc (Logic And Information Network Compiler) Reference Manual. Detroit Michigan, Burroughs Corporation.

Burstall, R., Collins, D., Popplestone, R. (1971) Programming In Pop-2. Edinburgh, Edinburgh University Press.

Cajori, F. (1974) A History Of Mathematical Notations. La Salle Illinois, Open Court. [Originally 1928]

Card, S. K., Moran, T.P., Newell, A. (1980) The Keystroke-Level Model For User Performance Time With Interactive Systems. Communications Of The ACM, 23:7, pp.396-410.

Card, S. K., Moran, T.P., Newell, A. (1982) Computer Text-Editing: An Information-Processing Analysis Of A Routine Cognitive Skill. Cognitive Psychology, 12, pp.396-410.

Carnevale, T. (1985) C To Pascal Conversion. Byte, 10:2. pp.139-144.

Carroll, J.M., Thomas, J.C. (1981) Human Factors In Communication. IBM Systems Journal, 20:2, pp.237-263.

Carroll, J.M., Thomas, J.C. (1982) Metaphor And The Cognitive Representation Of Computing Systems. IEEE Transactions On Systems, Man, And Cybernetics, 12:2, pp.107-116.

Cattel, R.G.G. (1980) Automatic Derivation Of Code Generators From Machine Descriptions. ACM Transactions On Programming Languages And Systems, 2:2, pp.173-190.

Chambers, J.M. (1982) Extending Ada Legally Via Preprocessors. Ada Letters, 1:4, pp.55-58.

Chang, S.K. (1987) Visual Programming Languages: A Tutorial And Survey. IEEE Software, 4:1, pp.29-39.

Chapin, N. (1970) Flow Charting With The Ansi Standard: A Tutorial. Computing Surveys, 2:2, p.119-146.

Cheriton, D.R., Malcolm, M.A., Melen, L.S., Sager, G.R. (1979) Thoth, A Portable Real-Time Operating System. Communications Of The ACM, 22:1, pp.105-114.

Cherry, L.L., Vesterman, W. (1980) Writing Tools - The Style And Diction Programs. In Leffler, Joy, and McKusick (Eds.) (1983).

Chomsky, N. (1965) Aspects Of The Theory Of Syntax. Cambridge Massachusetts, MIT Press.

Chomsky, N. (1968) Language And Mind. New York, Harcourt Brace Jovanovich.

Clark, R., Koehler, S. (1982) The UCSD Pascal Handbook. Englewood Cliffs New Jersey, Prentice-Hall.

Cleaveland, J.C., Uzgalis, R.C. (1977) Grammars For Programming Languages. New York, Elsevier North-Holland.

Clocksin, W., Mellish, C. (1981) Programming In Prolog. New York, Springer-Verlag.

Coelho H., Cotta, J.C., Pereira, L.M. (1980) How To Solve It With Prolog. Second Edition. Lisbon, Laboratoria Nacional De Engenharia Civil, Ministerio Da Habaticao E Obras Publicas.

Cohen, J. (1985) Describing Prolog By Its Interpretation And Compilation. Communications Of The ACM, 28:12, pp.1311-1324.

Cole, A.J. (1981) Macro Processors. Cambridge, Cambridge University Press.

Coleman, S.S., Poole, P.C., Waite, W.M. (1974) The Mobile Programming System Janus. Software: Practice And Experience, 4, pp.5.

Conway, M.E. (1958) Proposal For An Uncol. Communications Of The ACM, 1:10, pp.5-8.

Conway, M.E. (1963) Design Of A Separable Transition-Diagram Compiler. Communications Of The ACM, 6:7, pp.396-408.

Cooper D. (1983) Standard Pascal User Reference Manual. New York, W.W. Norton.

Coulter, N.S. (1983) Software Science And Cognitive Psychology. IEEE Transactions On Software Engineering, 9:2, pp.166-171.

Coulter, N.S., Kelly, N.H. (1986) Computer Instruction Set Usage By Programmers: An Empirical Investigation. Communications Of The ACM, 29:7, pp.643-647.

Cox, B.J. (1983) The Object-Oriented Pre-Compiler. ACM Sigplan Notices, 18:1, pp.15-22.

Cox, B.J. (1986) Object Oriented Programming: An Evolutionary Approach. Reading Massachusetts, Addison-Wesley.

Curtis, B. (1983) A Review of Human Factors Research On Programming Languages And Specifications. A.E.D.S. [Association For Educational Data Systems] Monitor, 21:9/10, pp.24-30.

Da Cruz, F., Tzoar, D., Catchings, B. (1983) Kermit Users Guide, Fourth Edition. New York, Columbia University Center for Computing Activities.

Dahl, O.-J., Nygaard, K. (1966) Simula - An Algol-based Simulation Language. Communications Of The ACM, 9:9, pp.671-681.

Dakin, R.J. (1975) A General Control Language: Language Structure And Translation. The Computer Journal, 18:4, p.324-332.

Darondeau, P., Le Guernic, P., Raynal, M. (1981) Types In A Mixed Language System. Bit, 21, pp.246-254.

Davis, P.J., Hersh, R. (1983) The Mathematical Experience. Harmondsworth Middlesex, Pelican. [Originally 1980]

De Millo, R.A., Lipton, R.J., Perlis, A.J. (1979) Social Processes And Proofs Of Theorems. Communications Of The ACM, 22:5, pp.271-280.

Denning, P.J. (1974) Is "Structured Programming" Any Longer The Right Term? ACM Sigplan Notices, 9:11, 4-5.

Denvir, B.T. (1979) On Orthogonality In Programming Languages. ACM Sigplan Notices, 14:7, pp.18-30.

Deutsch, P.L. (1981) Building Control Structures In The Smalltalk-80 System. Byte, 6:8, pp.323-346.

Deutsch, P.L., Lampson, B.W. (1967) An Online Editor. Communications Of The ACM, 10:12, pp.793-800.

Dijkstra, E.W. (1968a) Cooperating Sequential Processes. In F. Genuys (Ed.), Programming Languages. New York, Academic Press.

Dijkstra, E.W. (1968b) Go To Statement Considered Harmful. Communications Of The ACM. 11:3, pp.147-148.

Dijkstra, E.W. (1968c) The Structure Of The 'THE' Multiprogramming System. Communications Of The ACM, 11:5, pp.361-346.

Dijkstra, E.W. (1972) The Humble Programmer. Communications Of The ACM, 15:10, pp.860-866.

Dijkstra, E.W. (1975) Guarded Commands, Nondeterminacy, And Formal Derivation Of Programs. Communications Of The ACM, 18:8, pp.453-457.

Dijkstra, E.W. (1976) A Discipline Of Programming. Englewood Cliffs New Jersey, Prentice-Hall.

Dijkstra, E.W. (1980) My Hopes Of Computer Science. In Freeman, H, Lewis, P.M. II (Eds.) Software Engineering. New York, Academic Press.

Dijkstra, E.W. (1981) Why Correctness Must Be A Mathematical Concern. In Boyer, R.S., Moore, J.S. (Eds.) (1981) The Correctness Problem In Computer Science. London, Academic Press.

Dijkstra, E.W. (1983) Trip Report, Australia, 19 Jan. 1983 - 12 Feb. 1983. EWD 847. pp.0-6. [and related conversation]

Dreyfuss, H. (1955) Designing For People. New York, Hudson.

Dreyfus, H.L. (1979) What Computers Can't Do: The Limits Of Artificial Intelligence. Revised Edition. New York, Harper Colophon. [Originally 1972]

Einarsson, B., Gentleman, W.M. (1984) Mixed Language Programming. Software - Practice And Experience, 14, pp.383-395.

Elliot L.B., Holliday, F.L. (1987) Automating Conversion: Fortran To Ada With Ada. Computer Language, 4:7, pp.63-70.

Ellis, J.R. (1980) A Lisp Shell. ACM Sigplan Notices, 15:5, pp.24-34.

Falkoff, A.D., Iverson, K.E. (1978) The Evolution Of APL. ACM Sigplan Notices, 13:8, pp.47-58.

Farber, D.J., Griswold, R.E., Polansky, I.P. (1964) Snobol: A String Manipulation Language. Journal Of The ACM, 11:1, pp.21-30.

Fateman, R.J. (1982) High Level Language Implications Of The Proposed IEEE Floating Point Standard. ACM Transactions On Programming Languages And Systems, 4:2, pp.239-257.

Feldman, S.I. (1979) The Programming Language EFL. In Leffler, Joy, and McKusick (Eds.) (1983).

Feldman, S.I. (1986) Make - A Program For Maintaining Computer Programs. In McKusick, M.K., Karels, M.J., Bloom, J.M. (Eds.) (1986).

Feyerabend, P. (1978) Against Method. London, Verso.

Feuer, A.R., Gehani, N.H. (1979) A Comparison Of The Programming Languages C And Pascal - Part I: Language Concepts. Murray Hill New Jersey, Bell Laboratories (Internal Memorandum).

Feuer, A.R., Gehani, N.H. (1980) A Comparison Of The Programming Languages C And Pascal - Part I: Program Properties And Program Domains. Murray Hill New Jersey, Bell Laboratories (Internal Memorandum).

Feuer, A.R., Gehani, N.H. (1982) A Comparison Of The Programming Languages C And Pascal. ACM Computing Surveys, 14:1, pp.73-92.

Finzer, W., Gould, L. (1984) Programming By Rehearsal. Byte, 9:6, pp.187-210.

Fitzsimmons, A., Love, T. (1978) A Review And Evaluation Of Software Science. ACM Computing Surveys, 10:1, pp.3-18.

Floyd, R.W. (1979) The Paradigms Of Programming. Communications Of The ACM, 22:8, pp.455-460.

Foderaro, J.K. (1980) Franz Lisp Manual. In Leffler, Joy, and McKusick (Eds.) (1983).

Fraley, R.A. (1977) On Replacing Fortran. ACM Sigplan Notices, 12:9, pp.130-132.

Fraser, A.G. (1971) On The Meaning Of Names In Programming Systems. Communications Of The ACM, 14:6, pp.409-416.

Fraser, C.W., Hanson, D.R. (1983) A High Level Programming And Command Language. ACM Sigplan Notices, 18:6, pp.212-219.

Freak, R.A. (1981) A Fortran To Pascal Generator. Software - Practice And Experience, 11, pp.717-732.

Ganapathi, M., Fischer, C.N. (1982) Description-Driven Code Generation Using Attribute Grammars. ACM Sigplan Notices, 1982.

Ganapathi, M., Fischer, C.N. (1985) Affix Grammar Driven Code Generation. ACM Transactions On Programming Languages And Systems, 7:4, pp.560-599.

Ganapathi, M., Fischer, C.N., Hennessy, J.L. (1982) Retargetable Compiler Code Generation. ACM Computing Surveys, 14:4, pp.573-592.

Gannon, J.D. (1976) An Experiment For The Evaluation Of Language Features. International Journal Of Man-Machine Studies, 8:1, pp.61-73.

Gannon, J.D., Horning, J.J. (1975) Language Design For Programming Reliability. IEEE Transactions On Software Engineering, June, 1975.

Gannon, J.D., Katz, E.E., Basili, V.R. (1986) Metrics For Ada Packages: An Initial Study. Communications Of The ACM, 29:7, pp.616-623.

Gardner, H. (1976) The Shattered Mind. New York, Vintage Books.

Gazzaniga, M.S., LeDoux, J.E. (1978) The Integrated Mind. New York, Plenum.

Ghezzi, C., Jazayeri, M. (1982) Programming Language Concepts. New York, John Wiley & Sons.

Ghezzi, C., Mandrioli, D. (1979) Incremental Parsing. ACM Transactions On Programming Languages And Systems, 1:1, pp.58-70.

Gleitman, H., King Psammeticus' Experiment. In Psychology. New York, W.W. Norton.

Goedel, K. (1962) On Formally Undecidable Propositions. New York, Basic Books. [Originally 1931]

Goldberg, A., Robson, D. (1983) Smalltalk-80: The Language And Its Implementation. Reading Massachusetts, Addison-Wesley.

Goodman, D. (1987) The Complete Hypercard Handbook. New York, Bantam.

Goodman, N. (1978) Ways Of Worldmaking. Brighton, Harvester Press.

Goodman, N. (1981) Languages Of Art: An Approach To A Theory Of Symbols. Second Edition. Brighton, Harvester Press.

Goos, G., Wulf, W.A., Evans, E.Jr., Butler, K.J (Eds.) (1983) Diana: An Intermediate Language For Ada. Lecture Notes In Computer Science, 161, Berlin, Springer-Verlag.

Gordon, G. (1978) The Development Of The General Purpose Simulation System. ACM Sigplan Notices, 13:8, pp.183-198.

Graves, R.R. (1925) Poetic Unreason And Other Studies. London, Cecil Palmer.

Green, T.R.G., Sime, M.E., Fitter, M.J. (1981) The Art Of Notation. In Coombs, M.J., Alty, J.L. (Eds.) Computing Skills And The User Interface. London, Academic Press.

Gries, D. (1980) Current Ideas In Programming Methodology. In Wegner (Ed.) (1980). pp.254-275.

Gries, D. (1981) The Science Of Programming. New York, Springer-Verlag.

Griffiths, M. (1977) Translation Between High-Level Languages. In Brown (Ed.) (1977).

Griswold, R.E. (1972) The Macro Implementation Of Snobol 4. San Francisco, W.H. Freeman.

Griswold, R.E. (1978) A History Of The Snobol Programming Language. ACM Sigplan Notices, 13:8, pp.275-308.

Griswold, R.E., Griswold, M.T. (1983) The Icon Programming Language. Englewood Cliffs New Jersey, Prentice-Hall.

Griswold, R.E., Poage, J.F., Polansky, I.P. (1971) The Snobol4 Programming Language. Second Edition. Englewood Cliffs New Jersey, Prentice-Hall.

Hall, D.E., Scherrer, D.K., Sventek, J.S. (1980) A Virtual Operating System. Communications Of The ACM, 23:9, pp.495-500.

Halliday, M.A.K. (1978) Language As A Social Semiotic. London, Edward Arnold.

Hallpern, B. (1986) Multiparadigm Languages And Environments. IEEE Software, 86:1, pp.6-9,70-79.

Halstead, M.H. (1977) Elements Of Software Science. New York, Elsevier North-Holland.

Hammon, M., Landy, A. (1985) System Architecture Reference Guide, Revision 19.4. Natick Massachusetts, Prime Computer.

Hansen, W.J., Boom, H. (1978) The Report On The Standard Hardware Representation For Algol 68. Acta Informatica 9:2, ppp.105-119.

Hardy, S. (1984) A New Software Environment For List-Processing And Logic Programming. In O'Shea, T., Eisenstadt, M. (Eds.) Artificial Intelligence: Tools, Techniques, And Applications. Cambridge Massachusetts, Harper & Row.

Haring, G., Schechtner, O. (1983) On The Realization Of Extended Control Structures In Fortran. Software: Practice And Experience, 13, pp.431-446.

Harris, K. (1980) Forth Extensibility: Or How To Write A Compiler In 25 Words Or Less. Byte, 5:8, pp.164-184.

Harrison, M. (1973) Data Structures And Programming. Glenview Illinois, Scott Foresman.

Hart, H. (1982) Ada For Design: An Approach For Transitioning Industry Software Developers. Ada Letters, 2:1, pp.50-57.

Hartmanis, J., Stearns, R.E. (1965) On The Computational Complexity Of Algorithms. Transactions Of The American Mathematical Society, 117, pp.285-306.

Heering, J., Klint, P. (1985) Towards Monolingual Programming Environments. ACM Transactions On Programming Languages And Systems, 7:2, pp.183-213.

Henry, G.J. (1983) The Fair Share Scheduler. AT&T Bell Laboratories Technical Journal, 63:8, pp.1845-1857.

Hibbard, P.G. (1977) A Sublanguage of Algol 68. ACM Sigplan Notices, 12:5, pp.71-79.

Hill, I.D. (1972) Wouldn't It Be Nice If We Could Write Computer Programs In English - Or Would It? The Computer Bulletin, 16:6, pp.306-312.

Hill, I.D. (1983) Natural Language Versus Computer Language. In Sime and Coombs (1983). pp.55-72.

Hoare, C.A.R. (1974) Monitors: An Operating System Structuring Concept. Communications Of The ACM, 17:10, pp.549-557.

Hoare, C.A.R. (1978) Communicating Sequential Processes. Communications Of The ACM, 21:8, pp.666-677.

Hoare, C.A.R. (1981) The Emperor's Old Clothes. Communications Of The ACM, 24:2, pp.75-83.

Hobbs, J.R. (1977) What The Nature Of Natural Language Tells Us About How To Make Natural Language Like Programming More Natural. ACM Sigplan Notices 12:8, pp.85-93.

Hockey, S. (1980) A Guide To Computer Applications In The Humanities. London, Duckworth.

Hockey, S., Marriott, I. (1980) Oxford Concordance Program Users' Manual. Oxford, Oxford University Computing Service.

Hofstadter, D.R. (1980) Goedel, Escher, Bach: An Eternal Golden Braid. New York, Vintage Books.

Hofstadter, D.R. (1981) A Conversation With Einstein's Brain. In Hofstader, D.R, and Dennett, D.C. (Eds.) The Mind's I: Fantasies and Reflections on Self and Soul. Brighton, Harvester Press.

Holt, R.C., Wortman, D.B., Barnard, D.T., Cordy, J.R., (1977) SP/k: A System For Teaching Computer Programming. Communications Of The ACM, 20:5, pp.301-309.

Horak, W. (1985) Office Document Architecture And Office Document Interchange Formats: Current Status Of International Standardization. IEEE Computer, 18:12, pp.50-60.

Hume, A. (1980) Unix Scheduling. Australian Unix Users Group Newsletter, 2-2, p.24-30.

Hull, M.E.C. (1987) Occam - A Programming Language For Multi-Processor Systems. Computer Languages, 12:1, pp.27-38.

IBM (1964a) Operating System/360: Fortran IV, White Plains New York, International Business Machines.

IBM (1964b) Operating System/360: Job Control Language, White Plains New York, International Business Machines.

IBM (1965) Formac (Operating And User's Preliminary Reference Manual). Hawthorne New York, International Business Machines.

IBM (1967) PL/I - Formac Interpreter. Hawthorne New York, International Business Machines.

IBM (1972a) IBM/360 Operating System PL/I (F) Language Reference Manual. New York, International Business Machines.

IBM (1972b) SimPL/I General Information Manual. Croyden London, International Business Machines.

IBM (1976) OS PL/I Checkout And Optimizing Compilers: Language Reference Manual. New York, International Business Machines.

Ichbiah, J. (1984) Ada: Past, Present, And Future. Communications of the ACM, 27:10, pp.990-997.

Ichbiah, J.D., Heliard, J.C., Routine, O., Barnes, J.G.P., Krieg-Bruckner, Wichmann, B.A. (1979) Rationale For The Design Of The Ada Programming Language. ACM Sigplan Notices 14:6, part B.

IEEE (1981) [Computer Society Microprocessor Standards Committee Task P754] A Proposed Standard For Binary Floating Point Arithmetic. Draft 8.0. Computer, 14:3, pp.52-63.

IEEE (1986) [Portable Operating System For Computer Environments Committee] IEEE 1003.1 "Posix" Trial Use Standard [Book No. 967]. Los Angeles, IEEE Computer Society.

Ingalls, D.H.H. (1981) Design Principles Behind Smalltalk. Byte, 6:8, pp.286-298.

Inglis, J., King, P.J.H. (1977) Data Portability. In Brown (Ed.) (1977).

ISO (1980) Specification For The Computer Programming Language Pascal. International Standards Organisation.

Iverson, K.E. (1962) A Programming Language. New York, Wiley.

Iverson, K.E. (1980) Notation As A Tool Of Thought. Communications Of The ACM, 23:8, pp.444-465.

Jackson, M.A. (1975) Principles Of Program Design. New York, Academic Press.

Jackson, M.A. (1980) The Design And Use Of Conventional Programming Languages. In Smith, H.T., Green, T.R.G. (Eds.) Human Interaction With Computers. London, Academic Press.

James, J.S. (1980) What Is Forth? A Tutorial Introduction. Byte, 5:8, pp.100-126.

Jensen, K., Wirth, N. (1976) Pascal User Manual And Report. Second Edition. New York, Springer-Verlag.

Johnson, S.C. (1975) Yacc - Yet Another Compiler Compiler. Murray Hill New Jersey, Bell Telephone Laboratories.

Johnson, S.C. (1978) Lint, A C Program Checker. In Bell Telephone Laboratories (1979).

Johnson, S.C. (1980) Language Development Tools On The Unix System. Computer, 13:8, pp.16-24.

Johnson, S.C., Kernighan, B.W. (1973) The Programming Language B. Murray Hill New Jersey, Bell Telephone Laboratories.

Johnson, S.C., Lesk, M.E. (1978) Language Development Tools. Bell System Technical Journal, 57:6:2, pp.2155-2176.

Jones, D., Baskin, A.B., Chen, T., Bloomfield, L. (1979) Programs As Higher Level Subroutines. Software - Practice And Experience, 9, pp.149-156.

Joy, W. (1983) An Introduction To The C Shell. In Leffler, Joy, and McKusick (Eds.) (1983).

Joy, W., Cooper, E., Fabry, R., Leffler, S., McKusick, K., Mosher, D. (1983) 4.2BSD System Manual. In Leffler, Joy, and McKusick (Eds.) (1983).

Jung, C.G. (1972) Man And His Symbols. London, Aldus Press. [Originally 1964]

Kandinsky, W. (1947) Concerning The Spiritual In Art And Painting In Particular. New York, Wittenborn Schultz. [Originally 1912]

Karp, R.M. (1986) Combinatorics, Complexity, And Randomness. Communications Of The ACM, 29:2, pp.98-111.

Katzenelson, J. (1983) Introduction To Enhanced C. Software - Practice And Experience, 13, pp.551-596.

Kowalski, R.A., Sloman, A. et al. (1980) Panel Discussion On Knowledge Representation. ACM Sigart Newsletter, 70.

Kent, W. (1978) Data And Reality. Amsterdam, North-Holland.

Kernighan, B.W. (1975) Ratfor - A Preprocessor For A Rational Fortran. Software - Practice And Experience, 5, pp.395-406.

Kernighan, B.W. (1981) Why Pascal Is Not My Favorite Language. Murray Hill New Jersey, Bell Laboratories (Internal Memorandum).

Kernighan, B.W., Pike, R. (1983) The Unix Programming Environment. Englewood Cliffs New Jersey, Prentice-Hall.

Kernighan, B.W., Lesk, M.E., Ossana, J.F.Jr. (1978) Document Preparation. Bell System Technical Journal, 57:6:2, pp.2115-2136.

Kernighan, B.W., Plauger, P.J. (1974) The Elements Of Programming Style. New York, McGraw-Hill. [Second edition 1978].

Kernighan, B.W., Plauger, P.J. (1976) Software Tools. Reading Massachusetts, Addison-Wesley.

Kernighan, B.W., Plauger, P.J. (1981) Software Tools In Pascal. Reading Massachusetts, Addison-Wesley.

Kernighan, B.W., Ritchie, D.M. (1978) The C Programming Language. Englewood Cliffs New Jersey, Prentice-Hall.

Kilian, M.F. (1985) A Conditional Critical Region Pre-processor For C Based On The Owicki And Gries Scheme. ACM Sigplan Notices, 20:4, pp.42-51.

Knuth, D.E. (1968) The Art Of Computer Programming. Reading Massachusetts, Addison-Wesley.

Knuth, D.E. (1971) An Empirical Study Of Fortran Programs. Software - Practice And Experience, 1, pp.105-133.

Knuth, D.E. (1972) Ancient Babylonian Algorithms. Communications Of The ACM, 15:7, pp.671-677.

Knuth, D.E. (1974a) Computer Programming As An Art. Communications Of The ACM, 17:12, pp.667-673.

Knuth, D.E. (1974b) Structured Programming With Go To Statements. ACM Computing Surveys, 6:4, pp.261-301.

Knuth, D.E. (1979) Tex And MetaFont: New Directions In Typesetting. Bedford Massachusetts, American Mathematical Society & Digital Press.

Knuth, D.E. (1984) Literate Programming. The Computer Journal, 27:2, pp.97-111.

Krasner, G. (1983) Smalltalk-80: Bits Of History, Words Of Advice. Reading Massachusetts, Addison-Wesley.

Kuhn, T.S. (1970) The Structure Of Scientific Revolutions. Chicago, University Of Chicago Press. [Originally 1962]

La Palme, G., Chapleau, S. (1986) Logicon: An Integration Of Prolog Into Icon. Software: Practice And Experience, 16, pp.925-944.

Landin, P.J. (1966) The Next 700 Programming Languages. Communications Of The ACM, 9:3, pp.157-166.

Larmouth, J. (1973) Serious Fortran. Software - Practice And Experience, 3:2, pp.87-108, 3, p.197-226.

Larmouth, J. (1975) Scheduling For A Share Of The Machine. Software - Practice And Experience, 5, pp.29-49.

Larmouth, J. (1978) Scheduling For Immediate Turnaround. Software - Practice And Experience, 8, pp.559-578.

Lauder, P. (1980) Share Scheduling Works! Australian Unix Users Group Newsletter, 2-5, pp.11-14.

Layzell, P.J. (1985) The History Of Macro Processors In Programming Language Extensibility. Computer Journal, 28:1, pp.29-33.

Larson, M.L. (1984) Meaning Based Translation: A Guide To Cross-Language Equivalence. Lantham Maryland, United States University Press.

Lecarme, O.L. (1983) Review of Kernighan and Plauger (1981). ACM Computing Reviews, 24:5, pp.197-199.

Ledgard, H., Marcotty, M. (1975) A Genealogy Of Control Structures. Communications Of The ACM, 18:11, pp.629-639.

Ledgard, H., Marcotty, M. (1981) The Programming Language Landscape. Chicago, Science Research Associates.

Ledgard, H., Whiteside, J.A., Singer, A., Seymour, W. (1980) The Natural Language Of Interactive Systems. Communications Od The ACM, 23:10, pp.556-563.

Ledgard, H.F., Singer, A. (1982) Scaling Down Ada (Or Towards A Standard Subset). Communications Of The ACM, 25:2, pp.121-125.

Leffler, S.J., Joy, W.N., Fabry, R.S. (1983) 4.2BSD Networking Implementation Notes. In Leffler, Joy, and McKusick (Eds.) (1983).

Leffler, S.J., Joy, W.N., McKusick, M.K. (1983) Unix Programmer's Manual. Distribution 4.2. Berkeley California, Computer Science Division, Department of Electrical Engineering And Computer Science, University of California.

Lesk, M.E. (1975) Lex - A Lexical Analyser Generator. Murray Hill New Jersey, Bell Telephone Laboratories.

Leverett, B.W., Cattel, R.G.G., Hobbs, S.O., Newcomer, J.M., Reiner, A.H., Schatz, B.R., Wulf, W.A. (1980) An Overview Of The Production Quality Compiler-Compiler Project. Computer, 13:8, ppp.38-49.

Levin, J. (1980) Why A Lisp-Based Command Language? ACM Sigplan Notices, 15:5, pp.49-53.

Levis-Strauss, C. (1963) Structural Anthropology. New York, Elsevier. [Originally in French in 1958).

Lightfoot, D. (1982) The Language Lottery: Toward A Biology Of Grammars. Cambridge Massachusetts, MIT Press.

Limber, J. (1977) Language In Child And Chimp? American Psychologist, 32, pp.280-295.

Lindsay, C.H., And van der Meulen, S.G. (1981) Informal Introduction To Algol 68. Revised Edition. Amsterdam, North Holland.

Lions, J. (1983) Reflections On Unix. Australian Unix Users Group Newsletter, 3.

Lomow, G., Unger, B. (1984) Software Prototyping In Jade. In Jade Papers Presented At The Conference CIPS' 84. Calgary Alberta, Department Of Computer Science University Of Calgary.

Lycklama, H., Bayer, D.L. (1978) The Mert Operating System. The Bell System Technical Journal, 57:6:2, pp.2049-2086.

MacCallum, K.J., Schafe, L.T. (1974) A Mixed Language System: Pop 2 And Fortran. Software - Practice And Experience, 4, pp.145-154.

MacKenzie, C.E. (1980) Coded Character Sets, History And Development. Reading Massachusetts, Addison-Wesley.

Maclean, M.A., Peck, J.E.L. (1981) Chef: A Versatile Portable Text Editor. Software - Practice And Experience, 11, 467-477.

Maclean, M.A., Peck, J.E.L. (1982) The Chef Editor. Christchurch New Zealand, Computer Science Department University Of Canterbury.

MacLennan, B.J. (1979) Observations On The Differences Between Formulas And Sentences And Their Application To Programming Language Design. ACM Sigplan Notices, 14:7, pp.51-61.

MacLennan, B.J. (1983) Principles Of Programming Languages: Design, Evaluation, And Implementation. New York, CBS College Publishing.

Madsen, J. (1979) CCL - A High Level Command Language. Software - Practice And Experience, 9, pp.25-30.

Marcotty, M., Ledgard, H.F., Bochmann, G.V. (1976) A Sampler Of Formal Definitions. ACM Computing Surveys, 8:2, pp.191-276.

Marcotty, M., Sayward, F.G. (1977) The Definition Mechanism For Standard PL/I. IEEE Transactions On Software Engineering, 3:6, pp.416-450.

Marlin, C.D. (1976) An Experiment With The Extensibility Of Simula. ACM Sigplan Notices, 11:11, pp.50-57.

Martin, C.D. (1976) An Experiment With The Extensibility Of Simula. ACM Sigplan Notices, 11:11, pp.50-57.

Matthews, M.H. (1987) Prolog And C Join Forces, Computer Language, 4:7, pp.34-44.

Mayer, R.E. (1981) The Psychology Of How Novices Learn Computer Programming. Computer Surveys, 13:1, pp.121-141.

Mayer, R.E., Bayman, P. (1981) Psychology Of Calculator Languages: A Framework For Describing Differences In User's Knowledge. Communications Of The ACM, 24:8, pp.511-520.

McCarthy, J. (1978a) A Micro-Manual For Lisp - Not The Whole Truth. ACM Sigplan Notices, 13:8, pp.215-216.

McCarthy, J. (1978b) History of Lisp. ACM Sigplan Notices, 13:8, pp.217-223.

McCormack, J., Gleaves, R. (1983) Modula-2: A Worthy Successor To Pascal. Byte, 8:8, pp.385-395.

McGettrick, A.D. (1978) An Introduction To The Formal Definition Of Algol 68. Annual Review In Automatic Programming, 9, p1-84.

McGettrick, A.D. (1980) The Definition Of Programming Languages. Cambridge, Cambridge University Press.

McGraw, J.R., Andrews, G.R. (1979) Access Control In Parallel Programs. IEEE Transcations On Software Engineering, 5:1, pp.1-9.

McIlroy, M.D. (1960) Macro Instruction Extensions Of Compiler Languages. Communications Of The ACM, 3:4, pp.214-220.

McIlroy, M.D. (1972) The Roff Text Formatter. Murray Hill New Jersey, Bell Telephone Laboratories.

McLuhan, M. (1964) Understanding Media: The Extensions Of Man. New York, McGraw-Hill.

McKusick, M.K., Karels, M.J., Bloom, J.M. (Eds.) (1986) Berkeley Software Distribution 4.3 Unix Programmer's Manual. 6erkeley California, Computer Science Division, Department of Electrical Engineering And Computer Science, University of California.

McMahon, L.E. (1979) Sed - A Non-interactive Text Editor. In McKusick, M.K., Karels, M.J., Bloom, J.M. (Eds.) (1986)

Medina-Mora, R., Feiler, P.H. (1981) An Incremental Programming Environment. IEEE Transactions On Software Engineering, 7:5, pp.472-482.

Meeson, R., Pyster, A. (1979) Overhead In Fortran Preprocessors. Software - Practice And Experience, 9, pp.987-1000.

Meissner, L.P. (1975a) On Extending Fortran Structures To Facilitate SP. ACM Sigplan Notices, 10:9, pp.19-29.

Miller, A.I. (1989) Imagery In Scientific Thought. Cambridge Massachusetts, MIT Press.

Miller, G.A. (1956) The Magic Number Seven, Plus Or Minus Two: Some Limits On Our Capacity For Processing Information. Psychological Review, 63, pp.81-97.

Mohilner, P.R. (1977) Using Pascal In A Fortran Environment. Software - Practice And Experience, 7, pp.357-362.

Montgomery, E.B. (1982) Bringing Manual Input Into The 20th Century: New Keyboard Concepts. Computer, 15:3, pp.11-21.

Mooers, C.N. (1966) A Procedure-Describing Language For The Reactive Typewriter. Communications Of The ACM, 9:3, pp.215-219.

Moore, L. (1979) Design For Transportable Job Organisation Language. Computer Journal, 22:4, pp.296-302.

Morgan, H.L. (1970) Spelling Error Correction In System Programs. Communications Of The ACM, 13:2, pp.90-94.

Mulders, H. (1983) Some Observations On The In and Output In High Level Languages. ACM Sigplan Notices, 18:9, pp.55-58.

Muller, C. (1986) Modula-Prolog: A Software Development Tool. IEEE Software, 3:6, pp.39-45.

Murray, K.M.E. (1979) Caught In The Web Of Words - James Murray And The Oxford English Dictionary. Oxford, Oxford University Press.

Naur, P. (1975) Programming Languages, Natural Languages, And Mathematics. Communications Of The ACM, 18:12, pp.676-683.

Naur, P. (1978) The European Side Of The Last Phase Of The Development Of Algol 60. ACM Sigplan Notices, 13:8, pp.15-44.

Naur, P. (Ed.) (1963) Revised Report On The Algorithmic Language Algol 60. Communications Of The ACM, 6:1, pp.1-17.

Neal, R., Lomow, G., Peterson, M., Unger, B., Witten, I. (1984) Inter-Process Communication In A Distributed Programming Environment. In Jade Papers Presented At The Conference CIPS' 84. Calgary Alberta, Department Of Computer Science University Of Calgary.

Newy, M.C., Poole, P.C., Waite, W.M. (1972) Abstract Machine Modelling To Produce Portable Software - A Review And Evaluation. Software - Practice And Experience, 2.

Ng, N., Marsland, T.A. (1978) Introducing Graphics Capabilities To Several High Level Languages. Software - Practice And Experience, 8:5, pp.629-640.

Nicholls, J.E. (1975) The Structure And Design Of Programming Languages. Reading Massachusetts, Addison-Wesley.

Nie, N.H., Hull, C.H., Jenkins, J.G., Bent, D.H. (1975) Statistical Package For The Social Sciences. Second Edition. New York, McGraw-Hill.

Norman, D.A. (1981) The Truth About Unix. Datamation, 27:12, pp.138-150.

Nowitz, D.A., Lesk, M.E. (1979) A Dial-Up Network Of Unix Systems. Murray Hill New Jersey, Bell Telephone Laboratories.

Nygaard, K., Dahl O.-J. (1978) The Development Of The Simula Languages. ACM Sigplan Notices, 13:8, pp.245-272.

Obler, L.K., Albert, M.L. (1978) The Bilingual Brain: Neuropsychological And Neurolinguistic Aspects Of Bilingualism. New York, Academic Press.

Ossanna, J.F.Jr. (1976) Nroff/Troff User's Manual. In Bell Telephone Laboratories (1979).

Pagan, F.G. (1974) On Interpreter-Oriented Definitions Of Programming Languages. The Computer Journal, 19:2, pp.151-155.

Pagan, F.G. (1977) Algol 68 As An Implementation Language For Portable Interpreters. ACM Sigplan Notices, 12:6, pp.54-62.

Pagan, F.G. (1978) Algol 68 As A Metalanguage For Denotational Semantics. The Computer Journal, 22:1, pp.63-66.

Pagan, F.G. (1980a) Nested Sublanguages Of Algol 68 For Teaching Purposes. ACM Sigplan Notices, 15:7&8, pp.72-81.

Pagan, F.G. (1980b) On The Generation Of Compilers From Language Definitions. Information Processing Letters, 10:2. pp.104-107.

Pagan, F.G. (1982) Taming Ada For Introductory Teaching Purposes - An Approximation. Ada Letters, 1:4, pp.27-31.

Pagan, F.G. (1984) Toward Complete Programming Language Descriptions That Are Both Formal And Understandable. Software - Practice And Experience, 14, pp.199-206.

Pagan, P.G. (1981) Formal Specification Of Programming Languages: A Panoramic Primer. Englewood Cliffs New Jersey, Prentice-Hall.

Pakin, S. (1972) APL\360 Reference Manual. Second Edition. Chicago, Science Research Associates.

Palme, J. (1974) Simula As A Tool For Extensible Program Products. ACM Sigplan Notices, 9:2, pp.24-40.

Papanek, V. (1971) Design For The Real World. London, Thames And Hudson.

Parnas, D. (1972) On The Criteria To Be Used In Decomposing Systems Into Modules. Communications Of The ACM, 15:12, pp.1053-1058.

Parsons, I.T. (1975) A High Level Job Control Language. Software - Practice And Experience, 5, pp.69-82.

Pateman, T. (1981) Communicating With Computer Programs. Language And Communication, 1, pp.3-12.

Patrick, J.D. (1983) An Information Theoretic Interpretation Of Scientific Methodology, Australian Computer Science Communications, 5:1, pp.268-273.

Peck, J.E.L. (1982) A Tutorial Guide to Doris (A Text Formatting Program). Vancouver British Columbia, Department Of Computer Science, University Of British Columbia.

Pemberton, S., Daniels, M.C. (1982) Pascal Implementation (The P4 Compiler). Chicester, Ellis Horwood.

Pereira, F. (Ed.) (1983) CProlog User's Manual Version 1.2. Menlo Park California, SRI International.

Perkins, H. (1981) Lazy I/O Is Not The Answer. ACM Sigplan Notices, 16:4, pp.81-88.

Perlis, A. (1978) The American Side Of The Development Of Algol. ACM Sigplan Notices, 13:8, pp.3-14.

Perlman, G. (1984) Natural Artificial Languages: Low-Level Processes. International Journal Of Man-Machine Studies, 20:3, pp.1053-1058.

Plum, T. (1977) Fooling The User Of A Programming Language. Software: Practice And Experience, 7, pp.215-222.

Poole, L. (1985) The Excel Numbers Game. Macworld, 2:9, pp.87-91.

Portoghesi, P. (1983) Postmodernism, The Architecture Of The Post-Industrial Society. New York, Rizzoli.

Prenner, C.J. (1973) Extensible Control Structures. ACM Sigplan Notices, 8:9, pp.129-132.

Pritsker, A.A.B. (1974) The Gasp IV Simulation Language. New York, McGraw-Hill.

Project Jade (1984) Jade User's Manual. Calgary Alberta, Department Of Computer Science, University Of Calgary.

Prudom, A., Hennell, M.A. (1977) Fortran To Algol 68. ACM Sigplan Notices, 12:6, pp.138-143.

Pugh, J., Simpson, D. (1979) Pascal Errors - Empirical Evidence. Computer Bulletin, March, pp.26-28.

Pullen, D. (1964) A Fortran To Algol Translator. Computer Journal, 7:1, pp.24-27.

Pyle, I.C. (1979) Input/Output In High Level Programming Languages. Software: Practice And Experience, 9, pp.907-914.

Quarterman, J.S., Silbershatz, A., Peterson, J.L. (1985) 4.2BSD and 4.3BSD As Examples Of The Unix System. ACM Computing Surveys, 17:4, pp.379-419.

Quine, W.V.O. (1953) From A Logical Point Of View. Cambridge Massachusetts, Harvard University Press.

Quine, W.V.O. (1960) Word And Object. Cambridge Massachusetts, MIT Press.

Radin, G. (1978) The Early History Of PL/I. ACM Sigplan Notices, 13:8, pp.227-242.

Ramsdell, R. (1980) Power Of VisiCalc. Byte, 5:11, pp.190-192.

Rayner, D. (1975) Recent Developments In Machine Independent Job Control Languages. Software - Practice And Experience, 5, pp.375-394.

Redish, K.A., Smyth, W.F. (1986) Program Style Analysis: A Natural By-Product Of Program Compilation. Communications Of The ACM, 29:2, pp.126-133.

Rees, M.J. (1982) Automatic Assessment Aids For Pascal Programs. ACM Sigplan Notices, 17:10, pp.33-42.

Reps, T., Teitelbaum, T., Demers, A. (1983) Incremental Context-Dependent Analysis For Language-Based Editors. ACM Transactions On Programming Languages And Systems, 5:3, pp.449-477.

Richards, M. (1969) BCPL - A Tool For Compiler Writing. Proceedings Of The Spring Joint Computer Conference, 34, pp.557-566.

Richards, M. (1971) The Portability Of The BCPL Compiler. Software - Practice And Experience, 1, pp.135-146.

Richards, M., Aylward, A.R, Bond, P., Evans, R.D., Knight, B.J. (1979) Tripos - A Portable Operating System For Minicomputers. Software - Practice And Experience, 9, pp.513-526.

Richards, M., Whitby-Strevens, C. (1980) BCPL - The Language And Its Compiler. Cambridge, Cambridge University Press.

Ridler, P.F. (1979) A Fortran Reference Manual. London, Pitman.

Ripley, G.D., Druseikis, F.C. (1978) A Statistical Analysis Of Syntax Errors. Computer Languages, 3:4, pp.227-240.

Ritchie, D.M. (1978) Unix Time-Sharing System: A Retrospective. The Bell System Technical Journal, 57:6:2, pp.1947-1970.

Ritchie, D.M. (1983) A Stream Input-Output System. AT&T Bell Laboratories Technical Journal, 63:8, pp.1897-1910.

Ritchie, D.M. (1984) Reflections On Software Research. Communications of the ACM, 27:8, pp.758-760.

Ritchie, D.M., Johnson, S.C., Lesk, M.E., Kernighan, B.W. (1978) The C Programming Language. Bell System Technical Journal, 57:6:2, pp.1991-2020.

Ritchie, D.M., Thompson, K.L. (1974) The Unix Time-Sharing System. Communications of the ACM, 17:7, pp.365-375.

Robillard, P.N. (1986) Schematic Pseudocode For Program Constructs And Its Automation By Schemacode, Communications Of The ACM, 29:11, pp.1072-1089.

Robinson, I., (1975) The New Grammarians' Funeral: A Critique Of Noam Chomsky's Linguistics. Cambridge, Cambridge University Press.

Robinson, S.K., Torsun, I.S. (1977) The Automatic Measurement Of The Relative Merits Of Student Programs. ACM Sigplan Notices, 12:4, pp.80-93.

Rosenburg, H. (1975) Art On The Edge. Chicago, University of Chicago Press.

Ross, D.T. (1976) Homilies For Humble Standards. Communications Of The ACM, 19:11, pp.595-600.

Ryder, B.G. (1974) The PFort Verifier. Software - Practice And Experience, 4, pp.359-377.

Ryle, G. (1963) The Concept Of Mind. Harmondsworth Middlesex, Penguin. [Originally 1949]

Sabin, M.A. (1976) Portability - Some Experiences With Fortran. Software: Practice And Experience, 6, pp.393-396.

Sale, A.H.J. (1979a) Pascal Stylistics And Reserved Words. Software: Practice And Experience, 9, pp.821-826.

Sale, A.H.J. (1979b) Miniscules And Majuscules. Software: Practice And Experience, 9, pp.915-919.

Sale, A.H.J. (1981) Proposal For Extension To Pascal. ACM Sigplan Notices, 13:11, pp.98-103.

Saltzer, J.E. (1965) Runoff. In Crisman (Ed.) The Compatible Time-Sharing System. Cambridge Massachusetts, MIT Press.

Sammet, J.E. (1967) Roster Of Programming Languages For 1967. Computers And Automation, 16:6, pp80-82.

Sammet, J.E. (1969) Programming Languages: History And Fundamentals. Englewood Cliffs New Jersey, Prentice-Hall.

Sammet, J.E. (1972) Programming Languages: History And Future. Communications Of The ACM, 15:7, pp.601-610.

Sammet, J.E. (1978a) Roster Of Programming Languages For 1976/77. ACM Sigplan Notices, 13:11, pp.56-85.

Sammet, J.E. (1978b) The Early History Of Cobol. ACM Sigplan Notices, 13:8, pp.121-161.

Sammet, J.E. (1986) Why Ada Is Not Just Another Programming Language. Communications Of The ACM, 29:8, pp.722-733.

Sammet, J.E., Bond, E.R. (1964) Introduction To Formac. IEEE Transactions On Electronic Computers, 13:4, pp.386-94.

Sanden, B. (1985) Systems Programming With JSP: Example - A VDU Controller. Communications Of The ACM, 28:10, pp.1059-1067.

Sapir, E. (1931) Conceptual Categories In Primitive Languages. Science, 74, pp.578.

Saxe, J.B., Hisgen, A. (1978) Lazy Evaluation Of The File Buffer For Interactive I/O. Pascal News, 13, pp.93-94.

Schneiderman, B. (1980) Software Psychology: Human Factors In Computer And Information Systems. Cambridge Massachusetts, Winthrop.

Scott, D., Strachey, C. (1971) Towards A Mathematical Semantics For Computer Languages. Oxford, Oxford Computing Laboratory.

Scott, M.L. (1983) Messages Versus Remote Procedures Is A False Dichotomy. ACM Sigplan Notices, 18:5, pp.57-62.

Schwartz, B.M. (1972) Design And Implementation Of PL/I Pre-Processor Based Systems. ACM Sigplan Notices, 7:9, pp.21-36.

Sedelon, S.Y. (1970) The Computer In The Humanities And Fine Arts. ACM Computing Surveys, 2:2, pp.89-110.

Sethi R. (1981) Uniform Syntax For Type Expressions And Declarators. Software - Practice And Experience, 11, pp.623-628.

Seybold, J. (1985) Prime User's Guide, Revision 19.4. Natick Massachusetts, Prime Computer.

Shantz, P.W., German, P.W., Mitchell, J.G., Shirley, R.S.K., Zarnke, C.R. (1967) Watfor - The University Of Waterloo Fortran IV Compiler. Communications Of The ACM, 10:1, pp.41-44.

Sheil, B.A. (1981) The Psychological Study Of Programming. ACM Computing Surveys, 13:1, pp.101-120.

Shen, V.Y., Conte, S.D., Dunsmore, H.E. (1983) Software Science Revisited: A Critical Analysis Of The Theory And Its Empirical Support. IEEE Transactions On Software Engineering, 9:2, pp.155-165.

Shu, N.C., Housel, B.C., Lum, V.Y. (1975) Convert: A High Level Translation Definition Language For Data Conversion. Communications Of The ACM, 18:10, pp.557-567.

Shultis, J. (1983) A Functional Shell. ACM Sigplan Notices, 18:6, pp.202-211.

Sime, M.E., Green, T.R.G., Guest, D.J. (1973) Scope Marking In Computer Conditionals - A Psychological Evaluation. International Journal Of Man-Machine Studies, 5:2, 123-143.

Simpson, H.R., Jackson, K. (1979) Process Synchronisation In Mascot. Computer Journal, 22:4, pp.332-345.

Sintzoff, M. (1967) Existence Of A Van Wijngaarden Syntax For Every Recursively Enumerable Set. Annales De La Societe Scientifique De Bruxelles, 81, pp.115-118.

Skelly, P.G. (1982) The ACM Position On Standardization Of The Ada Language. Communications Of The ACM, 25:2, pp.118-120.

Slekys, A.G. (1979) Common Development Environment: Evaluation Phase Final Report. Toronto, Bell Northern Software Research.

Slekys, A.G., Delbarre, K.A., Hsu, G.F., Tilbrook, D.M. (1979) Common Development Environment: System Design Phase Final Report.

Sloman, A. (1978) The Computer Revolution In Philosophy: Philosophy, Science, And Models Of Mind. Hassocks Sussex, Harvester Press.

Sloman, A. (1985) Why We Need Many Knowledge Representation Formalisms. In Proceedings Of The 5th BCS Expert Systems Conference. London, British Computer Society.

Small, D.W., Weldon, L.J. (1983) An Experimental Comparison Of Natural And Structured Query Languages. Human Factors, 25, pp.253-263.

Smith, H.T., (1980) Human-Computer Communication. In Smith, H.T., Green, T.R.G. (Eds.) Human Interaction With Computers. London, Academic Press.

Snow, C.R. (1978) The Software Tools Project. Software - Practice And Experience, 8, pp.585-599.

Snyder, T. (1987) Puppy Love. Reading Massachusetts, Addison-Wesley.

Sprowls, R.C. (1972) PL/C: A Processor For PL/I. San Francisco, Canfield Press.

Stallman, R.M. (1986) GNU Emacs Manual. Cambridge Massachusetts, Free Software Foundation.

Standish, T.A. (1975) Extensibility In Programming Language Design. ACM Sigplan Notices, 10:7, pp.18-21.

Staunstrup, J. (1982) Message Passing Communication Versus Procedure Call Communication. Software - Practice And Experience, 12, pp.223-234.

Stoy, J.E. (1977) Denotational Semantics: The Scott-Strachey Approach To Programming Language Theory. Cambridge Massachusetts, MIT Press.

Strong, J., Olsztyn, T., Wegstein, J., Mock, O., Tritter, A., Steel, T., (1958) The Problem Of Programming Communication With Changing Machines: A Proposed Solution. Communications Of The ACM, 1:8, pp.12-18, 1:9, p.9-16.

Stroustrup, B. (1983) Adding Classes To The C Language: An Exercise In Language Evolution. Software - Practice And Experience, 13, pp.139-162.

Strunk, W., White, E.B. (1979) The Elements Of Style. Third Edition. New York, MacMillan. [Originally 1935]

Systems Research Laboratories (1985) SRL Artificial Intelligence Systems. Computer Systems Division, Systems Research Laboratories, Dayton, Ohio.

Takara, K. (1985) Programming Philosophy: Interviews With Donald Knuth and Niklaus Wirth. Computer Language, 2:5, p25-35.

Takeuchi, I, Okuno, H., Ohsato, N. (1983) Tao: A Harmonic Mean Of Lisp, Prolog, And Smalltalk. ACM Sigplan Notices, 18:7, pp.65-74.

Tanenbaum, A.S. (1981) Computer Networks. Englewood Cliffs New Jersey, Prentice-Hall.

Tanenbaum, A.S. (1987) Minix: A Unix Clone With Source Code For The IBM PC. Login: The Usenix Association Newsletter, 12:2, pp.3-9.

Tanenbaum, A.S. (1987) Operating Systems: Design And Implementation. Englewood Cliffs New Jersey, Prentice-Hall.

Tanenbaum, A.S., van Staveren, H., Keizer, E.G., Stevenson, J.W. (1983) A Practical Tool Kit For Making Portable Compilers. Communications Of The ACM, 26:9, pp.654-660.

Tanenbaum, A.S., van Staveren, J.W., Stevenson, J.W. (1982) Using Peephole Optimization On Intermediate Code. ACM Transactions On Programming Languages And Systems, 3:1, pp.21-36.

Teitelbaum, T., Reps, T. (1981) The Cornell Program Synthesizer: A Syntax Directed Programming Environment. Communications Of The ACM, 24:9, pp.563-573.

Tennant, R.D. (1977) Language Design Methods Based On Semantic Principles. Acta Informatica, 8:2, pp.97-112.

Thompson, K.L. (1984) Reflections On Trusting Trust. Communications of the ACM, 27:8, pp.761-763.

Trancz, W.J. (1979) Computer Programming And The Human Thought Process. Software: Practice And Experience, 9, pp. 127-138.

Triance, J.M., Layzell, P.J. (1985a) A Language Enhancement Facility For Cobol - Its Design And Implementation. Computer Journal, 28:1, pp.34-43.

Triance, J.M., Layzell, P.J. (1985b) Macro Processors For Enhancing High-Level Languages - Some Design Principles. Computer Journal, 28:2, pp.128-133.

Tsin, Y.H. (1982) Extending The Power Of Pascal's External Procedure Mechanism. Software - Practice And Experience, 12, pp.283-292.

Tucker, A.B. Jr. (1984) A Perspective On Machine Translation: Theory And Practice. Communications Of The ACM, 27:4, pp.322-329.

UK Ministry Of Defence (1979) The Official Handbook Of Mascot. Malvern England, Mascot Suppliers Association.

Unger, B., Birtwistle, G., Cleary, J., Hill, D., Lomow, G., Neal, R., Peterson, M., Witten, I.H., Wyvill, B. (1984) Jade: A Simulation And Software Prototyping Environment. In Bryant, R., Unger, B.W. (1984) Simulation In Strongly Typed Languages: Ada, Pascal, Simula ..., Proceedings Of The Conference On Simulation In Strongly Typed Languages. Society For Computer Simulation Series 13:2, pp.77-83.

USA Department Of Defense (1983) Military Standard Ada Programming Language. ANSI/MIL-STD-1815A, USA Department Of Defense.

USA Standards Institute (1966), USA Standard Fortran, ANSI Publication X3.9-1966.

USA Standards Institute (1969) Clarification Of Fortran Standards - Initial Progress. Communications Of The ACM, 12:5, pp.289-294.

USA Standards Institute (1971) Clarification Of Fortran Standards - Second Report. Communications Of The ACM, 14:10, pp.628-642.

van der Boos, J., Plasmijer, Hartel, P.H. (1983) Input Output Tools: A Language Facility For Interactive And Real-Time Systems. IEEE Transactions On Software Engineering, 9:3, pp.257-259.

van Wijngaarden, A. (1965) Orthogonal Design And Description Of A Formal Language, Amsterdam, Mathematisch Centrum.

van Wijngaarden, A. (Ed.) Mailloux, B.J., Peck, J.E.L., Koster, C.H.A. (1969) Report On The Algorithmic Language Algol 68. Numerische Mathematik, 14:2, pp.79-218.

van Wijngaarden, A., Mailloux, B.J., Peck, J.E.L., Koster, C.H.A., Sintzoff, M., Lindsey, C.H., Meertens, L.G.L.T., Fisker, R.G. (1976) Revised Report On The Algorithmic Language Algol 68. Acta Informatica, 5, pp.1-236.

Van Wyk, C.J. (1986) Awk As Glue For Programs. Software: Practice And Experience, 16, pp.369-388.

Venne, A., Fulchino, P. (1984) Primenet Guide, Revision 19.3. Framingham Massachusetts, Prime Computer. Vessey, I., Weber, R. (1986) Structured Tools And Conditional Logic: An Empirical Investigation. Communications Of The ACM, 29:1, pp.48-57.

Von Frisch, K. (1967) [Chadwick, C.E. (Tr.)] The Dance Language And Orientation Of Bees. Cambridge Massachusetts, Belknap Press.

Vouk, M.A. (1984) On The Cost Of Mixed Language Programming. ACM Sigplan Notices, 19:12, pp.54-60.

Waite, W.M. (1967) A Language Independent Macro Processor. Communications Of The ACM, 10:7, pp.433-440.

Waite, W.M. (1970a) Building A Mobile Programming System. Computer Journal, 13:1, pp.28-31.

Waite, W.M. (1970b) The Mobile Programming System: Stage2. Communications Of The ACM, 13:7, pp.415-421.

Wallis, B. (Ed.) (1984) Postmodernism - Addresses, Essays, Lectures - Art After Modernism: Rethinking Representation. New York, New Museum Of Contemporary Art.

Wallis, P.J.L. (1985) Automatic Language Conversion And Its Place In The Transition To Ada. In Ada In Use: Proceedings Of The Ada International Conference, Paris, Cambridge University Press. pp.275-284.

Wegner (1972) The Vienna Definition Language. ACM Computing Surveys, 4:1, pp.5-63.

Wegner, P. (Ed.) (1980) Research Directions In Software Technology. Cambridge Massachusetts, MIT Press.

Weinberg, G.M. (1970) PL/I: A Manual Of Style. New York, McGraw-Hill.

Weinberg, G.M. (1971) The Psychology Of Computer Programming. New York, Van Nostrand Reinhold.

Weissman, L. (1974) Psychological Complexity Of Computer Programs: An Experimental Methodology. ACM Sigplan Notices, 9:6, pp.25-36.

Weizenbaum, J. (1976) Computer Power And Human Reason: From Judgement To Calculation. San Francisco, W.H. Freeman.

Wetherell, C. (1977) Why Automatic Error Correctors Fail. Computer Languages, 2:4, pp.179-186.

Wetherell, C.S. (1980) Probabilistic Languages: A Review And Some Open Questions. ACM Computing Surveys, 12:4, pp.361-379.

Wexelblat, R.L. (1978) Preprints Of The ACM Sigplan History Of Programming Languages Conference. ACM Sigplan Notices, 13:8.

Wexelblat, R.L. (1981) The Consequences Of One's First Programming Language. Software - Practice And Experience, 11, pp.733-740.

Wexelblat, R.L. (Ed.) (1981) History Of Programming Languages. New York, Academic Press.

Whitaker, W.A. (1978) The US Department Of Defense Common High Order Language Effort. ACM Sigplan Notices, 13:2, pp.19-29.

Whitehead, A.N. (1911) An Introduction To Mathematics. Oxford, Oxford University Press.

Whorf, B.L. (1956) [Carroll, J.B. (Ed.)] Language, Thought, And Reality: Selected Writings Of Benjamin Lee Whorf. Second Edition. Cambridge Massachusetts, MIT Press.

Wichmann, B.A. (1984) Is Ada Too Big? A Designer Answers The Critics. Communications Of The ACM, 27:2, pp.98-103.

Wickens, C.D., Kramer, A. (1985) Engineering Psychology. Annual Review Of Psychology, 36, pp.307-348.

Wile, D., Balzer, R., Goldman, N. (1977) Automated Derivation of Program Control Structure From Natural Language Program Descriptions. ACM Sigplan Notices, 12:8, pp.77-84.

Wilkes, M.V. (1982) Hardware Support For Memory Protection: Capability Implementations. In Proceedings Of A Symposium On Architectural Support For Programming Languages And Operating Systems. ACM Sigplan Notices, 17:4, pp.108-116.

Williams, G. (1982) Lotus Development Corporation's 1-2-3. Byte, 7:12, pp.182-198.

Winograd, T. (1972) Understanding Natural Language. New York, Academic Press.

Winograd, T. (1979) Beyond Programming Languages. Communications Of The ACM, 22:7, pp.391-401.

Winograd, T. (1983) Language As A Cognitive Process. Cambridge Massachusetts, MIT Press.

Winograd, T. (1984) Computer Software For Working With Language. Scientific American, 251:3, pp.90-101.

Wirth, N. (1971a) Program Development By Stepwise Refinement. Communications Of The ACM, 14:4, pp.221-227.

Wirth, N. (1971b) The Programming Language Pascal. Acta Informatica, 1, 35-63.

Wirth, N. (1974) On The Composition Of Well-Structured Programs. ACM Computing Surveys, 6:4, pp.247-259.

Wirth, N. (1975) On The Design Of Programming Languages. In Information Processing 74: Proceedings of IFIP Congress 74. Amsterdam, North Holland.

Wirth, N. (1982) Programming In Modula-2. New York, Springer-Verlag.

Wirth, N. (1985) From Programming Language Design To Computer Construction. Communications Of The ACM, 28:2, pp.160-164.

Wirth, N., Hoare, C.A.R. (1966) A Contribution To The Development Of Algol. Communications Of The ACM, 9:6, 413-431.

Witten, I.H., Birtwistle, G.M., Cleary, J., Hill, D.R., Levison, D., Lomow, G., Neal, R., Peterson, M., Unger, B.W., Wyvill, B. (1983) Jade: A Distributed Software Prototyping Environment. ACM Operating Systems Review, 17:3, pp.10-23.

Wittgenstein, L. (1953) [Anscombe, G.E.M., Rhees, R. (Eds.) Anscombe, G.E.M. (Tr.)] Philosophical Investigations. Oxford, Basil Blackwell.

Wittgenstein, L. (1961) [Pears, D.F., McGuinness, B.F. (Tr.)] Tractatus Logico-Philosophicus. London, Routledge & Kegan Paul.

Wittgenstein, L. (1975) [Rhees, R. (Ed.), Hargreaves, R., White, R. (Trs.)] Philosophical Remarks. Oxford, Basil Blackwell. [Originally 1964]

Wolberg, J.R. (1981) Comparing The Cost Of Software Conversion To The Cost Of Programming. ACM Sigplan Notices, 16:4, pp.104-110.

Wolberg, J.R., Rafal, M. (1978) Convert - A Language For Program And Data File Conversions. Software - Practice And Experience, 8, pp.187-198.

Wooley, J.D. (1977) Fortran: A Comparison Of The Proposed New Language (1976) To The Old Standard (1966). ACM Sigplan Notices, 12:7, pp.112-125.

Wortman, D.B., Khaiat, P.J., Laskar, D.M. (1976) Six PL/I Compilers. Software - Practice And Experience, 6, pp.411-422.

Wright, P., Reid, F. (1973) Written Information: Some Alternatives To Prose For Expressing The Outcome Of Complex Contingencies. Journal Of Applied Psychology, 57, pp.160-166.

Wulf, W.A. (1972) The Problem Of The Definition Of Subroutine Calling Conventions. ACM Sigplan Notices, 7:12, pp.3-8.

Zelkowitz, M.V., Lyle, J.R. (1981) Implementation Of Language Enhancements. Computer Languages, 6:3/4, pp.139-153.

Zemanek, H. (1976) Some Philosophical Aspects Of Information Processing. In Zemanek, H. (Ed.) The Skyline Of Information Processing. Amsterdam, North-Holland.

REFERENCES FOR FREQUENTLY MENTIONED PROGRAMMING LANGUAGES AND SYSTEMS

APL	(1962) Iverson, K.E. (1972) Pakin, S. (1978) Falkoff, A.D., Iverson, K.E.
Ada	(1978) Whitaker, W.A. (1979) Ichbiah, J.D. et al. (1983) USA Department Of Defense
Algol	(1963) Naur, P. (Ed.) (1978) Naur, P. (1978) Perlis, A.
Algol 68	(1969) van Wijngaarden, A. (Ed.) et al. (1976) van Wijngaarden, A. et. al. (1981) Lindsay, C.H., And van der Meulen, S.G.
BCPL	(1969) Richards, M. (1980) Richards, M., Whitby-Strevens, C.
C	(1978) Kernighan, B.W., Ritchie, D.M.
Cobol	(1978b) Sammet, J.E.
Fortran	(1964a) IBM (1966) USA Standards Institute (1977) Wooley, J.D. (1978b) Backus, J.
Lisp	(1978a) McCarthy, J. (1978b) McCarthy, J.
Modula-2	(1982) Wirth, N.
PL/I	(1970) Weinberg, G.M. (1976) Ansi (1976) IBM (1978) Radin, G.
Pascal	(1971b) Wirth, N. (1976) Jensen, K., Wirth, N. (1980) ISO (1983) Cooper D.
Simula	(1966) Dahl, O.-J., Nygaard, K. (1973) Birtwistle, G.M., Dahl O.-J., Myrhaug, B. (1978) Nygaard, K., Dahl O.-J.
Smalltalk	(1983) Goldberg, A., Robson, D. (1983) Krasner, G.
Snobol	(1964) Farber, D.J., Griswold, R.E., Polansky, I.P. (1971) Griswold, R.E., Poage, J.F., Polansky, I.P. (1978) Griswold, R.E.
Unix	(1974) Ritchie, D.M., Thompson, K.L. (1985) Quarterman, J.S., Silbershatz, A., Peterson, J.L. (1986) Bach, M.J.

APPENDICES

NAME

IPIO - An Interprogramme Communication I/O Library

SYNOPSIS

```
ip -p pipetag -p pipetag ... command arg arg ...

#include <sys/file.h>

ipcreat(name, mode)
creat(name, mode)
char *name;

ipopen(path, flags, mode)
open(path, flags, mode)
char *path;
int flags, mode;

ipclose(d)
close(d)
int d;

cc = ipread(d, buf, nbytes)
cc = read(d, buf, nbytes)
int cc, d;
char *buf;
int nbytes;

cc = ipwrite(d, buf, nbytes)
cc = write(d, buf, nbytes)
int cc, d;
char *buf;
int nbytes;

ipioctl(d, request, argp)
ioctl(d, request, argp)
int d;
unsigned long request;
char *argp;
```

DESCRIPTION

Ipio allows interprogramme communications in a way portable across all Unix variants. Ordinary pipes are used, and are associated with "tag" names kept as environment variables. Programmes using the approach can communicate with each other using the file name oriented system calls with matching tag names, and other i/o system calls with consequent file descriptors.

The programme "ip" establishes pipes for each "-p" argument, with the tag name given. There are no inherent restrictions on tag names, but sh(1) expects all environment variable names to conform to its own rules (names should begin with a letter, and continue with letters, digits, or underscores) and because many programmes use sh indirectly, it's sensible to stick to those rules always. All names are internally prefixed with "IP" to reduce the possibility of clashing

with other software. Any subsequent arguments to "ip" are taken as arguments to exec, and current search rules are applied. If no arguments are given, a shell (with reference to SHELL environment variable - see below) is executed. Any descendent may then communicate with any other descendent using the functions and specifying the tag names.

All functions take arguments exactly as the system calls of the analogous names, and if not referring to a tagged pipe act exactly as the normal system calls. Where the given pathname is that of a tagged pipe, "creat" and "open" return a "dup" of the file descriptor, so that programmes with expectations of exact descriptor numbers - like csh(1) - will not be disappointed.

Tagged pipe descriptors are in danger from intermediate programmes - again like csh - that do much descriptor management. To reduce this danger, "ip" always uses descriptors from the top of the available range, furthest from any conventional usage. A "close" on any such primary tagged descriptor will be ignored.

Where "close" is done on a descriptor that has been in use as a tagged pipe, it is arranged that a "read" will interpret this as end-of-file. The effect is achieved by writing a special sequence through the pipe, but the same sequence passed in a buffer to "write" will pass transparently. The end-of-file condition is otherwise difficult to detect because of other programmes holding identical open descriptors, and the true condition is not generated until the last of these is closed.

The functions "read" and "write" maintain a record granularity orientation internally. Calls to "write" are regarded as defining records, and calls to "read" will return as much of the current or next record as the given buffer allows. "Ioctl" arranges for a tagged pipe to appear an interactive device, so disabling buffering otherwise done like Stdio(3).

There are two sets of these functions, those with names exactly the same as the system calls, and those prefixed "ip". The former may be loaded with a programme to effect the ordinary system calls being intercepted. The latter may be loaded where such direct interference with the system calls is to be avoided. All programmes wishing to communicate using these facilities, or even simply pass them on to programmes in descending processes, must use these functions, but most will simply need re-loading. Re-loaded versions of sh(1) and csh(1) are already available. When "ip" is given no programme to exec, it consults the SHELL environment variable, but uses the re-loaded versions instead of /bin/sh and /bin/csh. The shells are especially important because most programmes run in processes that descend directly from them, but they too may use the tagged pipes directly, eg.:

```
$ ip -p tube
$ cat /tmp/rhubarb >tube &
$ cat <tube >/tmp/crumble
```

would connect the output of the first "cat" to the input of the second.

BSD IPC

In BSD Unix, the Inter-Process Communication facilities (IPC) can be used to allow an identically working set of functions that do not require the preliminary programme "ip", or any inheritance scheme. In this way, programmes can be written so they are portable to other Unix environments, but can also take advantage of BSD facilities by ignoring the need for "ip".

Instead, any time a pipe tag name is given, a socket of type SOCK_STREAM (same semantics as an ordinary pipe) is created in the PF_UNIX protocol family and domain. An attempt is made to then connect(2) to an existing socket of the name, and if that fails because no such socket exists, then a bind(2) is done to create one, a listen(2) done to await another programme attempting connection, and a subsequent accept(2) done. If the bind fails, a final connect is attempted in case there was a race condition between two processes both using bind. Without the "ip" programme, tagged pipes must be distinguished in other ways: any name the same as an environment variable prefixed "IP" will be regarded a pipe tag, as will any name prefixed "=" (the symbol used by ls(1) to distinguish sockets).

Socket names in the UNIX domain do use the file system name space for coordination and access rights, and all sockets here use "/tmp". Functions using these BSD facilities are available both with system call names, and the same names prefixed "ip", in separate libraries to the functions described above.

FILES

/usr/local/ip
- the "ip" command.

/usr/local/ipsh /usr/local/ipio/ipcsh
- sh(1) and csh(1) with functions re-loaded.

/usr/local/lib/ipio/ipsyslib.o
- functions with system names.

/usr/local/lib/ipio/iplib.o
- functions with system names prefixed "ip".

/usr/local/ipio/ipbsh /usr/local/ipcio/ipbcsh
- sh(1) and csh(1) with bsd ipc functions re-loaded.

/usr/local/lib/ipio/ipbsyslib.o
- functions using bsd ipc with system names

/usr/local/lib/ipio/ipblib.o -
functions using bsd ipc, names prefixed "ip".

SEE ALSO

open(2), creat(2), close(2), read(2), write(2), ioctl(2), dup(2), pipe(2), socket(2), connect(2), bind(2), listen(2), accept(2), stdio(3).

BUGS

Really anti-social programmes can evade interception by accessing the *true* system calls by assembler, or by forcing interference on descriptors or environment variables.

AUTHOR

Robert Biddle.

NAME

Mux - A Language Multiplexor

SYNOPSIS

Mux [-vmcr] -[xXsSMDUILR string]* file file file ...

DESCRIPTION

Mux is a simple programming language multiplexor. The programme itself, however, is technically a file demultiplexor: it extracts separate components from files multiplexed in the appropriate format. Unlike programmes like ar(1), tar(1), or shar(1), where the internal format is a hidden implementation concern, the format and manipulation of the multiplexed file is a central concern of Mux. Also, there is no reverse action, no "multiplexing" provided automatically: by design that is something left to users. Like a macro-processor, Mux allows the laying out of programmes and related material in ways differing from the rules of any involved languages, yet providing the mechanism for automatic conversion to those rules. In particular, it is concerned with the ordering and inter-weaving of sections programmes and other files. For customising the design to a particular purpose, a macro processor would also well be used.

Mux allows for the division of a file into sectors, and for the selection and ordering of such sectors together. Sectors are labelled, and are selected by label, and both sectors and selections may specify files to be produced. In this way, a Mux file contains the information to construct other files according to the file specifications.

FORMAT

{<sector label file file file ... * comments... >}
...
{<sector>}

Sectors are distinguished by a "sector" directive, as above. They are labelled explicitly, and continue textually until a closing null sector directive, or another labelled sector directive - whereupon the sectors are nested. At the beginning of the file is an implicit sector, by default with a null label and no files specified. Files specified in the sector directive govern what is produced by the sector. If files are specified, the sector applies to those files only. If no files are specified, the sector applies to no particular file, but to any files involved in selection.

{<select label file file file ... * comments... >}

Selections are distinguished by a "select" directive, specify a sector label, and imply the production of that sector. If there are several sectors of the same name, they are produced in their order in the file. Files specified by the select directive govern what is produced by the sector. If files are specified, at most those files are produced

from the sector, possibly limiting the files specified at the sector. If no files are specified, the selection applies to any files specified at the sector. Note that in matching file specification, comparison is purely textual - functionally identical pathnames may be distinguished as different files.

In order that files may be produced with careful control over format, Mux directives may begin anywhere on a line, span lines, and end anywhere on a line.

USAGE

The command is used to extract files from Mux files. Any files given are used together in order. If there are none, the standard input is used instead.

-s label specifies a sector to be selected initially. Several such may be given, and only those - and in the order given - and their selections will be produced. By default, the textually first sector is selected.
-S label specifies a sector label implicit in the textually first sector. By default it is the null label. If more than one such specification is given, the later ones are taken as filename specifications for the first sector.
-x filename specifies a file to be extracted. Several such may be given. By default any files produced though selections and sectors are extracted.
-X filename[=pathname] specifies that a file being extracted should in fact be directed to a different pathname. If the pathname component is missing, the file is directed to the standard output. Several may be so specified, but will be interleaved as produced.

The flexibility of Mux is much greater when a macro processor is also used. Accordingly, access is built into the Mux command. The M4(1) macro processor is applied to the input if -m is specified. Any string specified by -M string is passed to M4 before the input, and so may be used to define macros etc. from the command line, and implies the "-m". To limit interference with other languages, the M4 pre-defined macros are prefixed with an "at-sign" @, so "define(YEAR, 1987)" would be "@define(YEAR, 1987)". All M4 facilities, file inclusion, expression evaluation, etc. are available.

M4 is a usefully sophisticated macro processor, and is also applicable to many languages, designed with no particular favourite in mind. In the Unix environment however, it's often the C language pre-processor that's viewed as a general purpose tool. The C pre-processor is applied if -c is specified. It is also implied by it's options -D string or -D string=value, which define macros to 1 and the given value respectively; -U string which undefines certain built-in macros; -I which specifies directories to be searched for file inclusion; and -r, which is required for macro recursion to be allowed. Again to limit interference between languages, the pre-processor commands begin with the "at-sign" @ rather than the "number-sign" #. Also, the C pre-processor ordinarily eliminates C comments - this is disabled here. Fanatics who use both macro processors will find that M4 is applied first.

-L string and -R string set the strings distinguishing Mux directives. By default, the left is "{<" and the right ">}". Note that the syntax doesn't require left and right to be distinct, but it can be easier to read if they are balanced opposites.
-v causes a verbose commentary to be directed to the standard error stream.

SUGGESTIONS FOR USE

Mux is useful whenever the order or presentation and interweaving of different components of a programme is important. A simple example would be the re-ordering of a Pascal programme to place less important procedures and functions later in the programme. Because files may weave together in some sectors, Mux provides a general approach to the Web system of Knuth, where programme source code and documentation cohabit the same files; source to be extracted and compiled, documentation to be extracted and formatted - and vice-versa! More generally, Mux is useful whenever joining dissimilar notations, enabling points of logical meeting to be physically juxtaposed, so weaving between the different rules of the different notation systems.

FILES

/tmp/MUXMXXXXX - m4 prep file
/tmp/MUXBXXXXX - seekable base file

SEE ALSO

B. W. Kernighan and D. M. Ritchie, The M4 Macro Processor
B. W. Kernighan and D. M. Ritchie, The C Programming Language, Prentice-Hall, 1978
Knuth, D.E., Literate Programming. The Computer Journal, 27:2, pp.97-111, (1984).
m4(1), cc(1)

BUGS

Make(1) is less effective when programme components share files.
The filename lists should possibly incorporate some pattern matching scheme.

AUTHOR

Robert Biddle.

NAME

Lys - Lex, Yacc, Simplified

SYNOPSIS

lys [-s] [-f] file file ...

DESCRIPTION

Lys is an attempt to simplify writing programmes involving the lexical analyser generator Lex(1), and the parser generator Yacc(1). It does this with other "high-level" tools, using the file multiplexor Mux(1) to generate correctly ordered and interleaved files from the Lys input, the macro processor M4(1) to make hide the Mux syntax, and Make(1) to assemble the various pieces.

Lys input is essentially the middle or "rules" component of Yacc input. As with Yacc, the grammer is written as a set of production rules, with the possibility of C language code associated with each rule. Lys provides the following direct assistance:

INLINE PATTERNS: No indirect terminal symbols are needed, as Lex regular expressions can be specified directly in the Yacc grammar by MATCH(expr). Different mentions of the same expression are folded automatically.

STRING TABLE: All such specified tokens are returned with the matching string as a value. Such strings are actually entries in a symbol table built automatically, so space is conserved, and string compare by pointer is possible.

CHARACTER DEFAULTS: All other input is returned character by character, so characters can be matched literally in the grammar. Exceptions are "white space" characters - blanks, tabs, and newlines - which are ignored. This category can be redefined by using IGNORE(...) where the arguments will be taken as a replacement list - so a null argument results in no characters being ignored.

DEBUGGING: If the switch "-D" is specified and recognised on the programme command line (see command argument facility below), a commentary will be written automatically to the standard error stream. The commentary displays all lines as they are read in, and displays each token as it is generated. The token is also described by a number unique to each type of token, and by the pattern that matched it.

ERROR DIAGNOSIS: Without advance knowledge of the syntax, error diagnosis can only be rough. By default, Lys arranges for an error message to printed on the standard error stream with the last line read in. This facility can be overridden with RECOVER(...) where any argument text will replace the Yacc grammar rule normally used.

In addition, some run-time routines are present that prove

useful:

SYMBOL TABLE ROUTINES: These are general purpose look-up table routines, mapping a character string to a pointer by hashing. They are also used internally by the String Table mechanism described above. Tabnew returns a new table; tabput takes a string "name" and a pointer to a value; and tabget returns a pointer to a value (or null) when passed a "name" string. The type "TTABENT" is intended as a union of pointer types.

```
TTAB tabnew();
tabput(TTAB, char *, TABENT);
TTABENT tabget(TTAB, char *);
```

COMMAND ARGUMENT ROUTINES: Arguments following a "-" are specified as either single character (a "switch") or single character followed by a string (a "flag"). These are specified in the Lys file by SWITCHES(...) and FLAGS(...). Scanning is done when the programme is executed, and then argcount(c) returns the integer number of arguments of character c; strings from flags are put in argv format vectors, returned by arglist(c); the nth argument is argstring(c, n), and any part following an "=" is argvalue(c, n). If the command arguments given do not match the pattern of switches and flags indicated, a "usage" message is generated automatically, and the programme exits.

INPUT/OUPUT ROUTINES: Vectors of filenames in argv format (like those returned by "arglist") may be opened all at once by OPENRV(v, fv) and OPENWV(v, fv) for files to be read and written respectively. The v is the filename vector, and fv is a declared and returned vector of references to standard I/O files, positionally matching the names given in v. Any file that cannot be opened causes an error message printed and, after checking others, results in programme exit. Programme input by default is the standard input, "stdin", but can be changed with INFILE(f) where f is any Standard I/O file reference. The directive INTRAP(trap) can be used to intercept input characters before they are scanned. The function "trap" will be be called with each character read, and its return value used.

Some facilities are also provided to access other parts of the Lex/Yacc/C facilities:

DEF(...) is inserted in the Lex "definition" section.
LEX(...) is inserted in the Lex "rules" section.
PREC(...) is inserted in the Yacc "precedence" section.
EXT(...) is inserted in the C section, after definitions but before code.
INIT(...) is inserted in the C code for main(), before the call to yyparse which provides the exit value.

USAGE

The Lys command arranges for any input files (taken together) to be processed and a complete executable programme produced. Files should be suffixed with ".lys", and need not be so specified. The output programme is the name of the first file, with ".lys" removed. All the work of

transforming the Lys file to executable is supervised by Make, and as with Make the -s switch specifies silence. Normally the programme prints the typical Make commentary. The -f specifies that various intermediate files should be retained - ordinarily they are removed unless errors are detected.

FILES

Lys uses several files, and like Lex and Yacc, it uses non-unique names in the working directory. Where "name" is the root of the first input file:

lex.LYS	Lex skeleton file.
yacc.LYS	YACC skeleton file.
tok.LYS	Token file used to fold tokens.
lex.tok.LYS	Token include file for Lex.
yacc.tok.Lys	Token folding file for Yacc.
name.l	Lex specification file.
name.y	Yacc specification file.
lex.yy.c	Lex output.
y.tab.[co]	Yacc output.
name	Executable.

SEE ALSO

Yacc(1), Lex(1), M4(1), Mux(1), Make(1)

BUGS

Debugging can be difficult with Lex and Yacc anyway. While Lys does make things simpler to specify, and provides useful run-time assistance, the increased distance from the C code can impede syntax debugging. For this reason, Lys is still not something for the absolute novice. However, it is useful to the more experienced programmer, and could lead to something more easily used still.

AUTHOR

Robert Biddle.

NAME

Rap - Rapport Between Systems

SYNOPSIS

Rap file

DESCRIPTION

Rap is a very small language for batch-oriented communication between a Unix host and a remote system. It provides a simple set of statements allowing commands to be sent, replies to be examined, and appropriate actions taken. It must be able to login on the system (or equivalent) with which communication is wished exactly as if it was being done manually, with userid, password, etc. Rap itself has no knowledge of commands or protocols for any remote system.

The set of statements is:

- connect "system";
The <system> is the short form of the name for the remote system serial line, normally a "special file" in the directory "/dev". This statement instructs Rap to connect the Unix host machine to the remote system, if possible, in accordance with the tip(1) serial line lock mechanism.

- disconnect;
Disconnects the Unix host from the connected remote system.

- send "string";
The <string> (any C format string) is sent on the serial line to the remote system. Control characters are also recognised by preceding them with "^" (e.g. control-X is ^X), and octal numbers are recognised as \ddd where <ddd> are octal digits. This statement forms the basis of any Rap program.

- set "string" "integer";
The set statement allows the user to change the following global characteristics (<integer> gives the new value):
(a) timeout: seconds waited before giving up looking for an expected match (default 30 seconds).
(b) delay: slowing mechanism when sending to the remote system to crudely avoid congestion on the line. The <integer> here should be a small positive number (say 0-10) where 10 would delay approximately a tenth of a second. Default value is 0, i.e. no delay.
(c) transcript: <integer> in this case is either 0 for no transcript (the default), or 1 requesting transcript of the session. The transcript file, when produced, always goes to the file "rap.trans".
(d) debug: <integer> can be one of the following -
0: no debugging, (default value)
1: lists data explicitly sent and expected data received,
2: lists all data sent and received.
All debugging information is written to the standard error stream.

- chdir "string";
Change working directory on the Unix host to <string>.

- shell "string";
Passes <string> to the Shell to execute on the Unix host. Standard input and output may be redirected as normally done in Shell syntax, or left connected to the remote system. With this command, once synchronisation has been achieved with send and expect, local and remote programs can directly communicate. One such especially useful program is kermit(1) for doing file transfer.

- expect { option defaultoption }
option => "string": action
defaultoption => default: action
action => a Rap statement

At least one of (<option>, <defaultoption>) must be present; <option> may appear more than once, but <defaultoption> may only appear once, and it must be the last. The <string> is any Grep(1) regular expression expected to be received from the remote system; when it *is* received, the action part is carried out. If not, the next option (if any) will be checked; all choices are checked with each new character received from the remote system. If no option is satisfied at timeout, the <defaultoption> action part will be carried out if present, otherwise the current compound statement fails.
At the outermost level, failure results in the program being aborted.

- try integer statement
The <integer> is any positive number, usually a small one; the <statement> is any Rap statement. Try causes <statement> to be executed repeatedly until successful (whereupon control proceeds on to the next statement) or when it has tried up to <integer> number of times (whereupon execution of the current compound statement fails).
At the outermost level, failure results in the program being aborted.

A Rap statement is any of the above mentioned statements or a compound one: a sequence of statements enclosed between "{" and "}".

Rap is used simply by
rap filename
where <filename> is the file containing the Rap program; it may optionally have a suffix ".r". Rap produces a C program in the file "filename.c" which can be compiled with cc(1).

EXAMPLE
This very simple example just illustrates logging on to a system called "prime", typing a message, and logging out.

```
set "delay" 1;
connect "prime";
```

```
try 5 { send "sync\n";
        expect { "ER" : send "login myuserid\n";}
}
expect { "word": send "mypassword\n"; }
expect { "OK":   send "type Hello world!\n"; }
expect { "OK":   send "logout\n"; }
expect { "OK":   ;
      default:  send "logout\n"; }
```

SEE ALSO

tip(1), kermit(1), grep(1)

BUGS

It's not clear that it should be a program generator - perhaps an interpreter would be better.
It really should use the /etc/remote database used by tip (1) to establish line parity and other characteristics.

AUTHORS

Rob Allan, Warwick Irwin, Chong Kuet Fung, Leong Yang Hee, supervised by Robert Biddle.